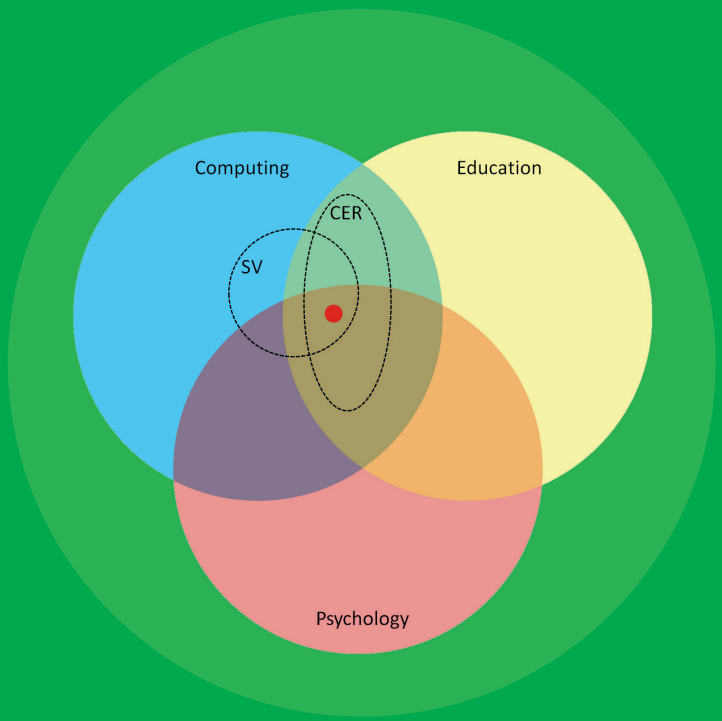
Department of Computer Science and Engineering

Visual Program Simulation in Introductory Programming Education

Juha Sorva



Aalto University

DOCTORAL DISSERTATIONS

Juha Sorva Visual Program Simulation in Introductory Programming Education Aalto University

Aalto University publication series
DOCTORAL DISSERTATIONS 61/2012

Visual Program Simulation in Introductory Programming Education

Juha Sorva

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 at the Aalto University School of Science (Espoo, Finland) on the 30th of May 2012 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering
Learning + Technology Group (LeTech)

Supervisor
Professor Lauri Malmi

Preliminary examiners
Professor Mark Guzdial, Georgia Institute of Technology, USA
Professor Emeritus Veijo Meisalo, University of Helsinki, Finland

Opponent
Associate Professor Mordechai Ben-Ari, Weizmann Institute of Science, Israel

Aalto University publication series
DOCTORAL DISSERTATIONS 61/2012

ISBN 978-952-60-4625-9 (printed)
ISBN 978-952-60-4626-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2012

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Juha Sorva

Name of the doctoral dissertation
Visual Program Simulation in Introductory Programming Education

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 61/2012

Field of research Computing Education Research

Manuscript submitted 27 February 2012 **Manuscript revised** 25 April 2012

Date of the defence 30 May 2012 **Language** English

☒ **Monograph** ☐ **Article dissertation (summary + original articles)**

Abstract

This thesis formulates and evaluates a pedagogical technique whose goal is to help beginners learn the basics of computer programming. The technique, visual program simulation (VPS), involves the learner in interactive simulations in which the learner takes on the role of the computer as the executor of a program. The student uses a given visualization of a so-called notional machine, an abstract computer, to illustrate what happens in memory as the computer processes the program. The purpose of these simulations is to help the beginner learn to reason about program execution, a skill whose development has been identified as a major challenge in introductory programming education. VPS promotes effective learning by seeking to cognitively engage the learner with a visualization. It can be made practical through visualization software. VPS software may also automatically assess students' simulations and provide personal feedback, which is a valuable asset especially in the large classes that are typical of introductory courses.

The thesis contributes to VPS in four ways. First, it formulates the concept of visual program simulation and outlines its underpinnings in terms of learning theory. Second, it presents a new software prototype that facilitates the use of VPS in practice. Third, it reports on a preliminary empirical evaluation of VPS and the software in the context of an introductory programming course. Fourth, it makes recommendations on the use of VPS in teaching and the further development of VPS tools, which arise from the empirical work.

The findings from a mixed-methods evaluation of VPS suggest that it is a promising pedagogical approach that helps many students learn programming. At the same time, the evaluation highlights certain important weaknesses. The purpose of VPS is not obvious to many students. Care must be taken to ensure that students develop a rich understanding of what VPS is and what they stand to gain from it. For best results, it is recommended that VPS be tightly integrated into the teaching and learning environment. The results from a controlled experiment further indicate that the short-term learning benefits of a VPS assignment are heavily dependent on which interactions the assignment demands from students. This implies that extreme care must be taken in the design of VPS systems and specific assignments so that required user interactions are aligned with intended learning goals.

On a more general level, the thesis serves as an example of educational tool development that is grounded in learning theory and informed by empirical evaluations. A fairly broad review of the literature on learning and teaching introductory programming is also contributed.

Keywords visual program simulation, introductory programming education, novice programmers, program visualization, interactive visualization for education

ISBN (printed) 978-952-60-4625-9 **ISBN (pdf)** 978-952-60-4626-6

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 428 **The dissertation can be read at** http://lib.tkk.fi/Diss/

A? Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Juha Sorva

Väitöskirjan nimi
Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 61/2012

Tutkimusala Tietotekniikan opetustutkimus

Käsikirjoituksen pvm 27.02.2012 **Korjatun käsikirjoituksen pvm** 25.04.2012

Väitöspäivä 30.05.2012 **Kieli** Englanti

☒ **Monografia** ☐ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Visuaalinen ohjelmasimulaatio (engl. visual program simulation, VPS) on tekniikka, jolla pyritään tukemaan aloittelijoita tietokoneohjelmoinnin oppimisessa. Siinä ohjelmoinnin oppija ottaa tietokoneen roolin ohjelman suorittajana. Hän käyttää vuorovaikutteista visualisaatiota abstraktista tietokoneesta kuvatakseen, mitä tietokone tekee, kun se käsittelee ohjelman askel askelelta. Tällaiset simulaatiot voivat opettaa aloittelijaa järkeilemään ohjelmien suoritusvaiheista ja näin selviytymään eräästä ohjelmoinnin opintojen varhaisvaiheen keskeisestä haasteesta. VPS pyrkii tehostamaan oppimista edellyttämällä opiskelijalta aktiivista visualisaation käyttöä sen sijaan, että tämä vain katselisi annettua kuvamateriaalia. Toimiakseen käytännössä VPS tarvitsee tuekseen tarkoitukseen laaditun apuohjelman. VPS-järjestelmän avulla voidaan myös automaattisesti arvioida opiskelijoiden suoriutumista ja tarjota henkilökohtaista palautetta. Tämä on arvokas etu erityisesti suurilla massakursseilla, jollaisia ohjelmoinnin johdantokurssit usein ovat.

Väitöskirja edistää VPS:ää neljällä tavalla. 1) Väitöskirjassa muotoillaan visuaalisen ohjelmasimulaation käsite ja sen oppimisteoreettinen pohja. 2) Väitöskirja esittelee uuden järjestelmäprototyypin, joka mahdollistaa VPS:n käytännön opetuksessa. 3) Väitöskirjassa raportoidaan tuloksia alustavista empiirisistä tutkimuksista, joissa arvioidaan VPS:ää ja mainittua järjestelmää erään ohjelmoinnin peruskurssin yhteydessä. 4) Näihin tuloksiin nojaten väitöskirjassa esitetään suosituksia siitä, miten VPS:ää kannattaa käyttää opetuksessa sekä siitä, millaisiksi VPS-järjestelmiä tulisi jatkossa kehittää.

Tutkimustapoja yhdistelevän arvioinnin tulokset viittaavat siihen, että VPS on lupaava opetustekniikka, joka auttaa monia opiskelijoita oppimaan ohjelmointia. Toisaalta arviointi nostaa esille myös tärkeitä heikkouksia. VPS:n merkitys ei ole opiskelijoille itsestäänselvää. Opetuksessa on pidettävä huoli siitä, että opiskelijoille muodostuu rikas ymmärrys siitä, mitä VPS on ja miten he voivat siitä hyötyä. Parhaat oppimistulokset saavutettaneen integroimalla VPS tiukasti muuhun oppimiskontekstiin. Esitetyt tutkimustulokset puoltavat näkemystä, jonka mukaan VPS:n lyhyen aikavälin oppimisvaikutukset riippuvat vahvasti siitä, mitkä asiat korostuvat tehtävän opiskelijalta vaatimassa vuorovaikutuksessa. Näinollen simulointitehtäviä ja -järjestelmiä suunniteltaessa on erinomaisen huolellisesti sovitettava yhteen tehtävien oppimistavoitteet ja simulaation vaatimat toimenpiteet.

Yleisemmällä tasolla väitöskirja toimii esimerkkinä oppimistyökalun kehittämishankkeesta, joka pohjautuu oppimisteoriaan ja kokemusperäiseen tutkimukseen. Väitöskirja sisältää laajan kirjallisuuskatsauksen ohjelmoinnin alkeiden oppimisesta ja opetuksesta.

Avainsanat visuaalinen ohjelmasimulaatio, ohjelmoinnin perusopetus, ohjelmoinnin aloittelijat, ohjelmavisualisaatio, vuorovaikutteinen opetusvisualisaatio

ISBN (painettu) 978-952-60-4625-9 **ISBN (pdf)** 978-952-60-4626-6

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2012

Sivumäärä 428 **Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/

Acknowledgements

Lauri Malmi first hired me as an inexperienced programming teacher, and some years later introduced me to computing education research. This thesis results from Lauri's recognition of the need for sound cross-disciplinary research on computing and engineering education, a goal towards which he has steered his research group in recent years. Thank you, Lauri, also for allowing me the freedom to pursue my interests, while nevertheless gently guiding this seemingly ever-growing project to completion.

Teemu Sirkiä's contribution to this thesis has been invaluable. It was with Teemu that we designed the UUhistle software, which Teemu has implemented single-handedly. It has been a great pleasure to work on this project together.

Anders Berglund introduced me to phenomenographic research and has gone to great trouble to help me experience it in ever richer ways. Anders's enthusiasm for computing education research is infectious, as is his eagerness to learn about and find applications for educational theory. In many ways, what I have learned through Anders has helped me understand what it means to be a researcher.

I almost certainly would not have written a thesis on visual program simulation if it were not for a conversation that I had with Ari Korhonen and Ville Karavirta back in 2007, in which we brainstormed ideas on the use of visualization in courses on computing. It was from this seed that this thesis eventually grew.

The research that I report in this book has been conducted in collaboration with several people. Jan Lönnberg, Ville Karavirta, Kimmo Kiiski, and Teemu Koskinen contributed to parts of this work. I am indebted to Kerttu Pollari-Malmi for allowing us to use her course as a guinea pig, and to her students for participating in our research.

Otto Seppälä, Juha Helminen, Päivi Kinnunen, and Petri Ihantola provided useful references, for which I am very grateful. I further thank all the members of the Learning+Technology research group for their feedback and for stimulating discussions on program visualization, computing education, the thesis manuscript, and what not.

I am grateful to the pre-examiners, Professor Mark Guzdial and Professor Emeritus Veijo Meisalo, for taking the time to read my work and for their valuable observations and suggestions concerning it.

I would like to thank the Koli Calling research community for providing a friendly platform for presenting and discussing ideas. And for the hotel room parties.

I gratefully acknowledge the indirect influence of Professor Jorma Sajaniemi on this work. Saja's inspiring work on the roles of variables catalyzed my interest in the psychology of programming.

I thank my parents, in particular, for their help during this busy time, and, in general, for everything.

I would like to mention that my kids, Rafa and Juju, are totally awesome.

Last, and most importantly, I thank my wife Rosana for her patience, love, and support. I'm sorry for all the late nights. Thank you. Smooch!

Contents

List of Figures

List of Tables

Chapter 1

Here is How to Make Sense of This Thesis

1.1 This work is about a pedagogical technique for improving programming education

In this thesis, I formulate and evaluate a pedagogical technique whose goal is to help beginners learn the basics of computer programming. This technique, *visual program simulation* (VPS for short), involves the learner in interactive simulations in which the learner assumes the role of the computer as executor of a program. Such simulations can help the beginner learn to reason about program execution, a skill whose development has been identified as a major challenge in introductory program education.

Visual program simulation promotes effective learning by seeking to cognitively engage the learner with a visualization of computer memory. It can be made practical through visualization software. Such software may also automatically assess students' simulations and provide feedback, which is a valuable asset especially in the large classes that are typical of introductory courses.

1.2 This is computing education research on software visualization

Denning et al. (1989) defined *computing* as "the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application." In other words, computing deals with the theory and practice of computers, including computer software and hardware. The work presented in this thesis relates to computing in the dual sense that it involves the design and use of a computing application – a software system that supports visual program simulation – for the purpose of learning about computing. Within computing research, the field of *software visualization* (SV) studies the creation and use of visual representations of computer software for various purposes; the present work is an example of educationally motivated research on software visualization.

This work also has significant elements of social science: I study the relationships between people and VPS, and the effects that VPS has on people as they learn. As dissertations for the degree of "Doctor of Science in Technology" go, mine is a relatively 'soft' one – although perhaps instead of hard and soft sciences we should speak of hard and difficult sciences, respectively, as rigorous research on human thought and behavior is anything but easy (cf., e.g., Diamond, 1987). My work falls in the domain of *computing education research* (CER), the multidisciplinary field that investigates the learning and teaching of computing. In addition to being a subfield of computing and education, CER draws on the theories and methods of psychology. Figure 1.1 illustrates these relationships.

The main application of computing education research is to help computing educators develop what Shulman (1986) called *pedagogical content knowledge* – knowledge of particular content from the point of view of teaching it – and *curricular knowledge* – knowledge of the various alternative approaches and techniques for teaching about a subject. Indirectly, the beneficiaries of computing education research include students learning about computing – an increasingly varied group – and, by extension, everyone whose life is or could be affected by the products of computing, that is, nearly everybody.

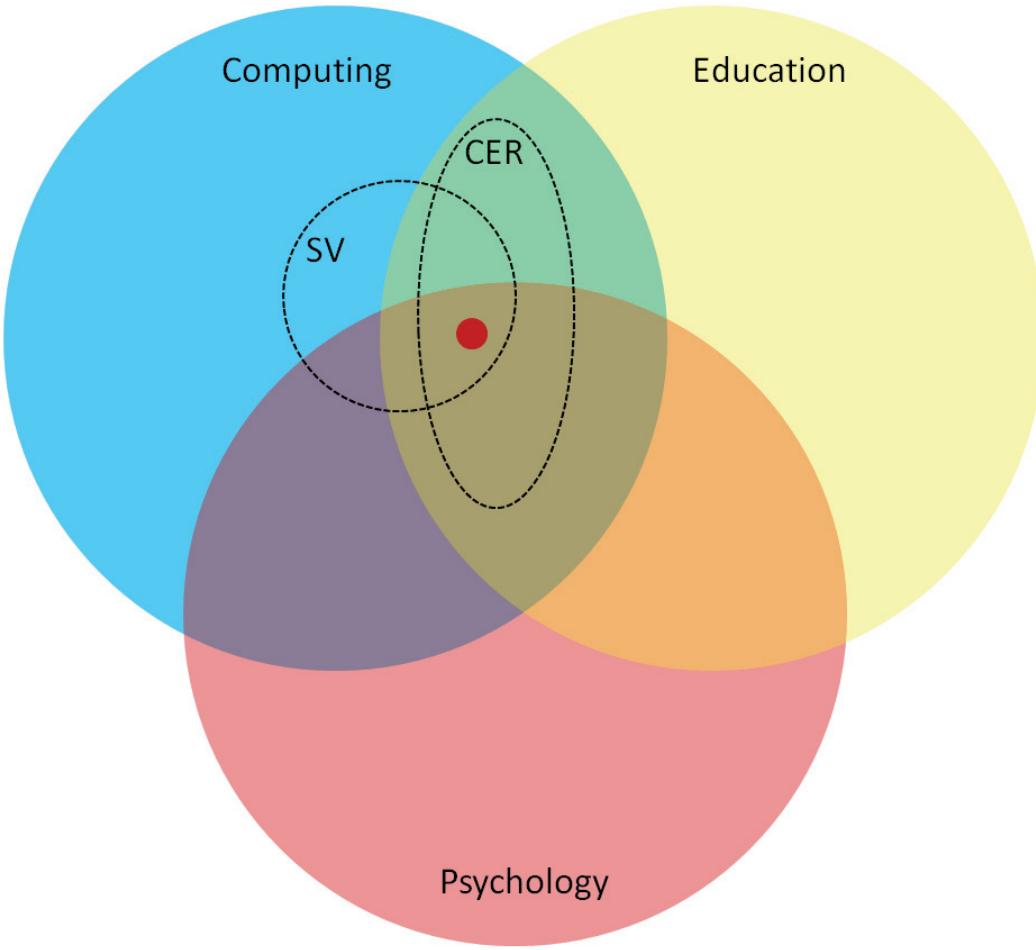


Figure 1.1: This thesis – the red dot – in the context of three disciplines of research: computing, education, and psychology. CER stands for computing education research, SV for software visualization. The size of each area is unimportant.

On research traditions

Each research tradition tends to think particularly highly of its own way of seeing and doing things. In the words of one of association football's erudite minds,

> *Everyone thinks they have the prettiest wife at home.* (Arsène Wenger, quoted by BBC Sport, 2004)

My relationship with research traditions, with theories and methods, is polyamorous. This thesis builds on multiple traditions – primarily schema theory and mental model theory in cognitive psychology, the phenomenographic tradition within educational research, and the software visualization tradition. Further influences come from the constructivist paradigm of education and educational research on threshold concepts. The influence of different traditions is reflected in the theoretical groundwork of visual program simulation, as well as in my empirical work, which I undertake from a pragmatic mixed-methods perspective.

1.3 This thesis consists of six main parts

The first three parts of this dissertation constitute a literature survey on the learning and teaching of programming. I review what has been written, and comment on it, in an attempt to synthesize a coherent picture of pertinent aspects of introductory programming education. Part I, *The Challenge of Introductory Programming Education*, establishes that there is a problem: introductory programming courses are not working nearly as well as educators – and students – would like. Part II, *Learning Introductory Programming*, considers learning to program from the perspective of several general theories of learning as well as prior research within CER. Part III, *Teaching Introductory Programming*, reviews approaches to teaching introductory programming courses and software visualization tools designed to help with this task.

Part IV, *Introducing Visual Program Simulation*, presents visual program simulation as a pedagogical technique, and our particular implementation of it in a software system. Part V, *Empirical Investigations of Visual Program Simulation*, moves us from constructive research to evaluative empirical research as I report the results from a set of interrelated studies in which my colleagues and I explored the use of VPS in the context of an introductory programming course. Part VI, *Conclusions*, looks back at what has been achieved, and forward to future work.

Each of these parts is prefaced with an introduction that provides an overview of the upcoming chapters.

Part VII is just for the end bits: appendices and bibliography.

1.3.1 The thesis is driven by a sequence of questions and answers

The present work can be expressed as a sequence of questions and answers. The first chapters chart the terrain by putting questions to the existing literature. The answers provoke more questions, which produce more answers. Eventually, the answers motivate the formulation of visual program simulation in Part IV, and lead to the research questions posed to empirical data in Part V. Figure 1.2 summarizes some of the main contents of this book from a question-setting perspective.

For a more traditional exposition of my research questions for empirical research, see Chapter 16.

1.3.2 There are different ways of reading the thesis

I expect that most readers of this thesis will have some expertise in computing. Throughout this work, I refer to fundamental programming concepts, common programming languages, and programming paradigms under the assumption that they will be familiar to the reader. However, I do cover in some detail even the well-known theories from education and psychology that I build on. This is to help readers that do not have a background in these other fields. The lengthier treatment of educational and psychological theory also reflects my own learning process. Writing about these topics has been a device for learning as I have approached these other disciplines during my postgraduate studies in computing.

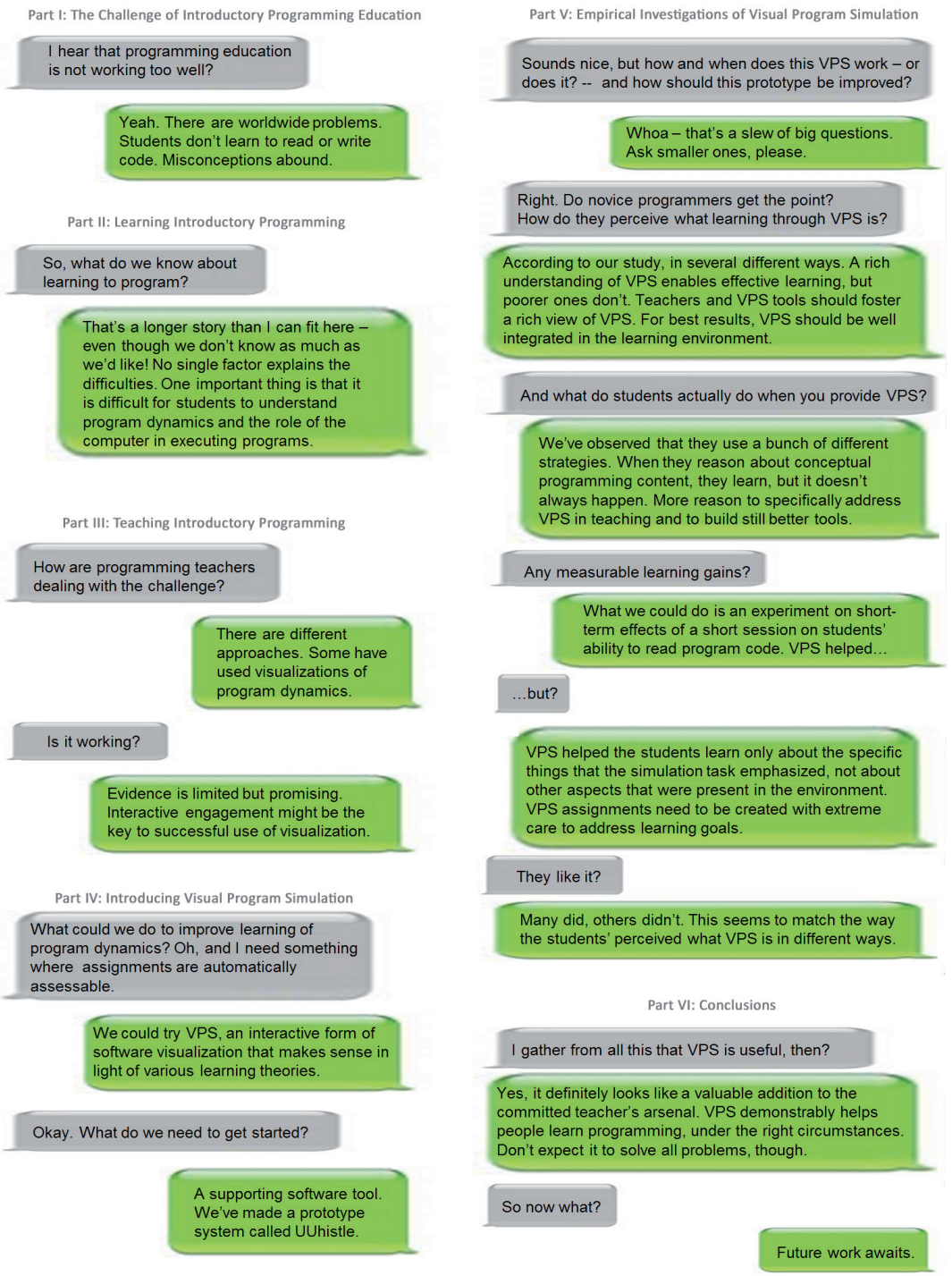


Figure 1.2: This thesis as dialogue.

The thesis should ideally be read from beginning to end. There are threads of thought that are gradually developed throughout the book. However, readers with different backgrounds and goals may wish to do something different.

Readers with a strong background in computing education research may wish to skip what is familiar in Parts I to III, and focus on Parts IV to VI, which deal directly with visual program simulation.

The reader in a hurry to find out something about VPS may wish to look at just the first two chapters (12 and 13) of Part IV, and the conclusions in Chapter 21.

The table of contents is designed to read like an extended abstract of sorts; some readers may find this useful for obtaining a quick overview.

Or, if you are one of my relatives with a limited interest in computing education, or otherwise stuck at my doctoral defense with a copy of this book in front of you, you may prefer to try the surface approach provided by the crossword in Figure 1.3. (This one might take a while to complete.)

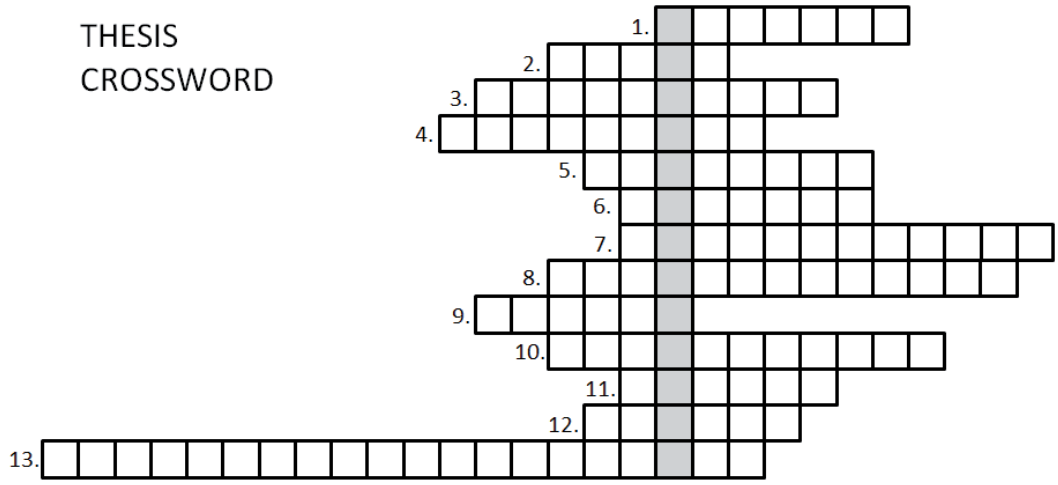


Figure 1.3: 1. The most common word of five or more letters in this thesis.
2. The first name of the supervisor of the thesis.
3. The key to the success of educational visualization?
4. A type of cognitive load.
5. Describing the machine that students control in visual program simulation.
6. The most common surname within the list of references.
7. An abstraction of validity and credibility.
8. "Assignment moves a value from a variable to another", for one.
9. Describing the kind of mental model suitable for fixing problems.
10. What threshold concepts tend to be for learners.
11. The more abstract of UUhistle's parents.
12. Any word with A as its third letter might be considered to be, as an answer to this question?
13. The longest unhyphenated English word in this thesis.
Down: The kind of understanding of this thesis that this crossword is likely to engender.

Part I

The Challenge of Introductory Programming Education

Introduction to Part I

Introductory programming courses around the world are failing to teach students how to program. While many students certainly do succeed, too many others fall far short of the goals set by curriculum planners and teachers. Evidence from computing education research shows that the problem is significant and not just local to a few institutions.

Programming is an activity that is central to the field of computing. This is evidenced in computing education by the dominance of programming-first approaches to teaching introductory computing courses (ACM and IEEE Computer Society, 2001). The primary goal of a typical introductory computing course – commonly called a *CS1 course* or simply *CS1* – is that students learn to create programs in some programming language. Unfortunately, it turns out that this is a demanding goal for an introductory course, and one that is often not met. In Part I, I try to substantiate these claims.

Part I consists of two chapters. Chapter 2 examines the goals of programming education from the viewpoint of two influential educational taxonomies, Bloom's taxonomy and SOLO. Chapter 3 is about the unwelcome evidence: many students do not acquire even rudimentary programming skills in CS1, and the problem is widespread. These chapters set the scene for a more detailed look at what it takes to learn to program, which will follow in Part II.

Chapter 2

Introductory Programming Courses are Demanding

This chapter takes a look at the goals of introductory programming education in the light of two educational taxonomies: Bloom's taxonomy and the SOLO taxonomy. These taxonomies enable us to characterize programming tasks in terms of their cognitive and structural complexity, which gives us an idea of what is commonly expected of beginner programmers.

In Section 2.1 below, I introduce Bloom's taxonomy and its applications to programming. Section 2.2 deals with SOLO.

2.1 Bloom's taxonomy sorts learning objectives by cognitive complexity

Some learning objectives are harder to achieve than others. It is much harder to learn to evaluate the quality of computer programs than to list programming keywords, for instance. In the 1950s, a group of educators led by Benjamin Bloom defined a taxonomy that divides learning into three domains: the cognitive, affective, and psychomotor (Bloom, 1956). In the same work, they further detailed the cognitive branch of the taxonomy by presenting a hierarchy of learning objectives ranked according to their expected cognitive complexity (see Figure 2.1). Bloom's work has received wide acclaim and remains highly influential.

The name "Bloom's taxonomy" is used in two different ways. It sometimes refers to the overall taxonomy, with cognitive, affective, and psychomotor branches. It is also commonly used to refer to the taxonomy of learning objectives within the cognitive domain. In this thesis, "Bloom's taxonomy" is shorthand for "Bloom's taxonomy of learning objectives for the cognitive domain".

2.1.1 The taxonomy distinguishes between six types of cognitive process

The six levels of the original 1956 taxonomy, from lowest to highest, are:

1. Knowledge: the student can recall specific facts or methods. This level is characterized by verbs such as enumerate, name, and define;

2. Comprehension: the student understands the meaning of facts or concepts. This level is characterized by verbs such as explain, discuss, and paraphrase;

3. Application: the student can solve problems by applying knowledge to new concrete situations. This level is characterized by verbs such as produce, implement, and solve;

4. Analysis: the student can break down information into its parts to determine motives or causes, or to make inferences. This level is characterized by verbs such as analyze, discriminate, and infer;

5. Synthesis: the student can combine elements in new ways to produce novel wholes. This level is characterized by verbs such as create, compose, and invent;

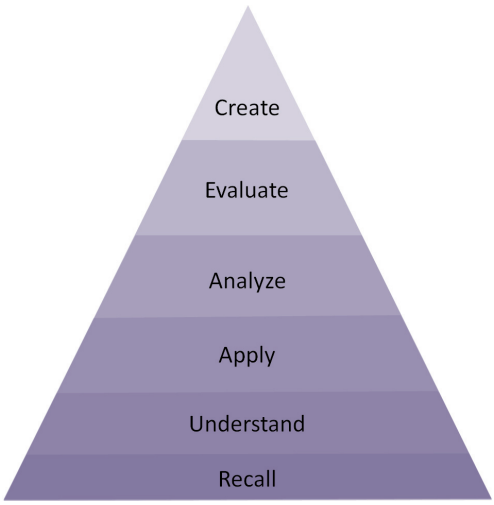


Figure 2.1: Bloom's taxonomy of learning objectives for the cognitive domain (the level names shown are from the revised taxonomy by Anderson et al., 2001).

6. Evaluation: the student can make judgments about material in light of selected criteria. This level is characterized by verbs such as appraise, critique, and compare.

Many variants of the taxonomy have been proposed in the literature. An influential variant – which I will refer to as the *revised Bloom's taxonomy* – was defined by an interdisciplinary group of experts led by Anderson and Krathwohl (Anderson et al., 2001). The revised Bloom's taxonomy has two dimensions – a cognitive process dimension similar to that of the original taxonomy, and a knowledge dimension that specifies the type of content being processed – factual, conceptual, procedural, or metacognitive. Anderson et al. also exchanged the places of the two last levels of the cognitive process dimension to produce a revised hierarchy: recall, understand, apply, analyze, evaluate, and create, as shown in Figure 2.1.

2.1.2 Applying Bloom to programming is tricky but increasingly popular

In comparison to its esteemed status in other subfields of education, Bloom's taxonomy had received relatively little attention in programming education until recent years. However, in the past decade or so, a growing body of work has emerged that relates the taxonomy – either the original or the revised version – to introductory programming. Many programming educators have reported on how they have used Bloom's taxonomy to motivate improvements to the instruction or assessment of programming courses (e.g., Buck and Stucki, 2000; Lister and Leaney, 2003; Scott, 2003; Thompson et al., 2008; Starr et al., 2008; Khairuddin and Hashim, 2008; Alaoutinen and Smolander, 2010). The recent CER literature also features a thread that applies Bloom's taxonomy to study the learning of programming and discuss the appropriateness of forms of assessment in CS1 (e.g., Johnson and Fuller, 2006; Fuller et al., 2007; Whalley et al., 2006, 2007; Meerbaum-Salant et al., 2010). The revised Bloom's taxonomy is being used as a guideline by ACM and IEEE's work on developing their computer science curriculum (ACM and IEEE Computer Society, 2008). Several variants of Bloom's taxonomy have been proposed to be particularly suitable for programming education (Shneider and Gladkikh, 2006; Fuller et al., 2007; Bower, 2008).

No general consensus has emerged on precisely how to map the goals of programming education onto Bloom's taxonomy. Code-tracing skills, for instance, have been variously classified within the literature as understand or analyze, and many interpretations have been presented as to how to 'Bloom rate' program-writing assignments of different kinds. Gluga et al. (2011) found that academics untrained in use of Bloom's taxonomy for the classification of programming assignments in a particular way produced a variety of different classifications.

There is some convergence of opinion, too. The ability to create a program to solve an unfamiliar problem, the literature widely agrees, belongs at the synthesis level in the original taxonomy, or create in the revised version. Scholars within CER, as well as Bloom's original group, have stressed the relevance of students' prior knowledge in determining the cognitive demands of an activity (e.g., Johnson and Fuller, 2006; Thompson et al., 2008; Gluga et al., 2011). For instance, Thompson et al. (2008) consider that applying programming knowledge involves solving familiar problems with new data or solving unfamiliar problems that match a familiar pattern or require an algorithm that is known to the student. To come up with a previously unknown kind of solution is to create.

Despite these challenges of interpretation, Bloom's taxonomy can tell us something about CS1 courses:

2.1.3 The goals of introductory programming courses are cognitively challenging

Bloom's taxonomy is intended to be used by teachers as a tool for analyzing and designing courses and curricula. In particular, the taxonomy was created to emphasize that learning objectives should not be set only at the lowest levels, as was being done in many traditional educational settings, but at all levels of the taxonomy. David Krathwohl, a member of both Bloom's original group and the one that revised it decades later, looks back on how Bloom's taxonomy has been applied across disciplines:

> *One of the most frequent uses of the original Taxonomy has been to classify curricular objectives and test items in order to show the breadth, or lack of breadth, of the objectives and items across the spectrum of categories. Almost always, these analyses have shown a heavy emphasis on objectives requiring only recognition or recall of information, objectives that fall in the Knowledge category. But it is objectives* [...] *in the categories from Comprehension to Synthesis that are usually considered the most important goals of education. Such analyses, therefore, have repeatedly provided a basis for moving curricula and tests toward objectives that would be classified in the more complex categories.* (Krathwohl, 2002)

It is interesting to observe that in the field of computer programming, applying Bloom's taxonomy to course evaluation has not precipitated the kind of shift that Krathwohl describes, from knowledge towards more complex educational goals. In fact, what effect there has been, has been largely in the opposite direction. Applying Bloom's taxonomy to introductory programming education has highlighted the fact that even introductory courses set the 'cognitive bar' very high for would-be programmers.

Lister and Leaney (2003) note that the traditional problem-solving goal of a CS1 course – to be capable of developing a (small) program which solves a given problem that has been expressed vaguely in non-programming terms – corresponds to synthesis (or create) high up in the taxonomy. Typical introductory programming exams emphasize writing code, that is, application and/or synthesis, depending on the question and the interpretation of Bloom (Scott, 2003; Petersen et al., 2011; Simon et al., 2012).

Oliver et al. (2004) examined computing courses in terms of their 'Bloom rating' (a weighted average of the numbered Bloom levels of each type of assessment). They discovered that programming courses, including introductory ones, have high Bloom ratings, whereas courses on other computer science topics had much lower ratings.[1]

The findings of Whalley et al. (2006) suggest that the higher up in the revised Bloom's taxonomy a programming task is, the more difficult it is for students succeed in it – as Bloom's group hypothesized more generally.

These Bloom-driven analyses show the goals of a typical introductory programming course to be quite demanding. Another influential taxonomy, SOLO, lends weight to this conclusion.

[1] Although Oliver et al. (2004) analyzed only computing courses, one suspects that typical introductory courses in many non-computing subjects also tend to have significantly lower 'Bloom ratings' than introductory programming courses.

2.2 The SOLO taxonomy sorts learning outcomes by structural complexity

Some educators have questioned the appropriateness of Bloom's taxonomy for the design of learning activities and assessments.

> *When using Bloom's taxonomy, the supposition is that the question leads to the particular type of Bloom response. There is no necessary relationship, however, as a student may respond with a very deep response to the supposedly lower order question: "Describe the subject matter of Guernica?" Similarly, a student may provide a very surface response to "What is your opinion of Picasso's Guernica?"* (Hattie and Purdie, 1998, p. 161)

One solution to such mismatches between activity and outcome is to categorize outcomes. This is the focus of the *Structure of the Observed Learning Outcome*, or *SOLO*, a taxonomy formulated by Biggs and Collis (1982) from empirical analyses of students' responses to learning tasks.

2.2.1 SOLO charts a learning path from disjointed to increasingly integrated knowledge

SOLO's five levels can be used to categorize learner responses in terms of their structural complexity. Paraphrased from Biggs and Tang (2007, pp. 77-78), the levels of SOLO are:

1. Prestructural: a response at this level misses the point or consists of empty phrases, which may be elaborate but show little evidence of actual learning;
2. Unistructural: this kind of response meets only a single part of a given task or answers only one aspect of question. It misses other important attributes entirely;
3. Multistructural: the response is 'a bunch of facts'. It expresses knowledge of various important aspects, but does not connect them except possibly on a surface level. The learner sees 'the trees' but not 'the forest';
4. Relational: the response relates and integrates facts into a larger whole that has a meaning of its own. It is no longer a list of details; rather, facts are used by the learner to make a point;
5. Extended abstract: a response at this level goes beyond what is given and applies it to a broader domain.

SOLO describes a systematic progression in performance as an individual learns. First, from the prestructural through to the multistructural level, the learner makes quantitative progress, increasing the amount of knowledge they have. Progressing to the relational and extended abstract levels involves a qualitative change as meaning emerges from the increasingly well perceived connections between elements of knowledge (Figure 2.2).

SOLO is intended to be used by teachers both for analyzing responses to learning activities (e.g., answers to questions) and for setting learning objectives. SOLO and Bloom's taxonomy have somewhat different perspectives – Bloom classifies learning objectives (skills), while SOLO classifies learning outcomes (responses to activities) – and they are, to a certain extent, complementary. Both can be used to characterize the objectives set for learners.

2.2.2 SOLO has been used to analyze programming assignments

A research project called BRACElet has recently applied SOLO to code-reading and code-writing tasks in the context of introductory programming (Lister et al., 2006b; Sheard et al., 2008; Clear et al., 2009; Lister et al., 2009a; Whalley et al., 2011). They present an interpretation of how SOLO applies to simple code comprehension problems of the form "in plain English, explain what the following segment of Java code does", and characterize the four main levels as follows (Lister et al., 2009a):

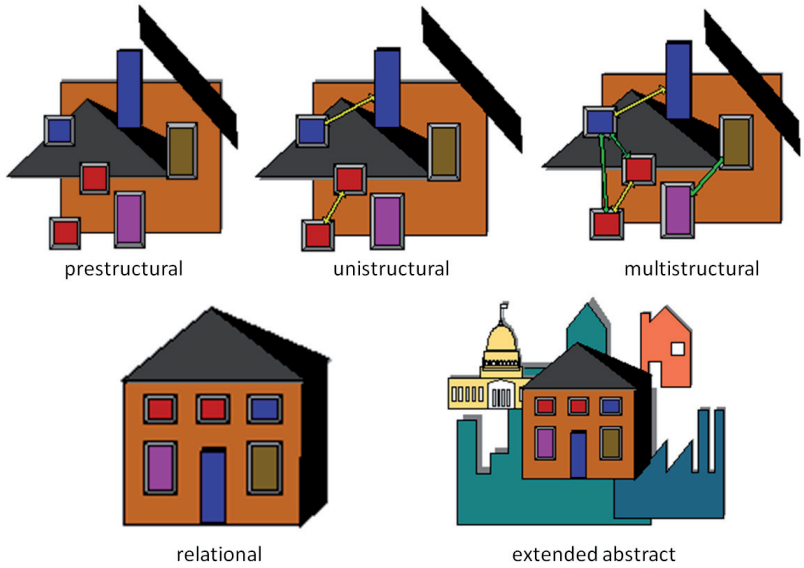


Figure 2.2: Atherton's (n.d.) metaphorical illustration of SOLO's five levels. At the unistructural and multistructural levels, links between previously unconnected pieces are increasingly perceived. A qualitative change happens at the relational level, at which point the whole becomes genuinely meaningful.

1. Prestructural: substantially lacks knowledge of programming constructs or is unrelated to the question;

2. Unistructural: a description of one part of the code;

3. Multistructural: a line-by-line description of all the code (the 'trees');

4. Relational: a summary of what the code does in terms of its purpose (the 'forest').

The results of Sheard et al. (2008) suggest that the degree of structuredness of students' responses to a code-reading task measured on such a SOLO-based scale correlates significantly and positively with their ability to write program code.

As for writing code, one suggestion for categorizing responses was sketched out by Clear et al. (2009); I paraphrase:

1. Prestructural: inability to write correct code;

2. Unistructural: ability to write a single small piece of code, e.g., to increment the value of a variable.

3. Multistructural: ability to write combine a few statements to write a multi-line solution based on a detailed specification or pseudocode. E.g., completing a method so that it returns false if a given book is on loan but true otherwise;

4. Relational: ability to write code to solve a problem which has not been specified to the extent that the problem represents pseudocode for the solution. E.g., writing a class to represent library books.

Like Bloom's taxonomy, SOLO provides us with a lens through which to examine the goals of introductory programming education.

2.2.3 The expected outcomes of programming courses are structurally complex

In terms of the SOLO taxonomy, successfully writing programs requires an understanding of programs that reaches the relational level. Program design tasks may even require students to transfer programming concepts beyond what they have encountered or learned about and to the extended abstract level. In a vein of work similar to the 'Bloom rating' measurements of Oliver et al. (Section 2.1.3 above), Brabrand and Dahl (2009) analyzed the stated requirements of the computer science, mathematics, and natural science courses of a Danish university where all teachers are required to use the SOLO taxonomy as they specify course goals. They found that computer science courses in general had significantly higher SOLO levels than natural science courses and (even more clearly) than mathematics courses. Typical programming-related competencies desired were relational – at a high level.

Both Bloom's taxonomy and SOLO illuminate the challenge of introductory programming education: the learning objectives are cognitively challenging, the expected learning outcomes structurally complex. The emphasis on synthesis skills and relational knowledge in introductory programming education is not surprising in light of the history of programming education. Early programming courses introduced programming as a tool for practitioners of other sciences rather than as a facet of computer science (ACM and IEEE Computer Society, 2001, p. 22). The goal was that these practitioners – a different demographic group than today's CS1 students – would take perhaps only a single course in programming and would then be able to apply what they learned to difficult problems within their main field.

There is no question that programming continues to be a key skill within computing, and is an ever more important tool for non-computer-scientists. Developing learners' ability to create novel programs will continue to be the central goal of programming education. Nevertheless, analyses of the cognitive demands of introductory programming courses have led researchers to ring a warning bell (Whalley et al., 2006):

> *It appears likely that programming educators may be systemically underestimating the cognitive difficulty in their instruments for assessing programming skills of novice programmers. For non-elite institutions it is likely that some proportion of the high failure rate in introductory programming may be attributed to this difficulty in setting fair and appropriate assessment instruments.* [...] *The level of difficulty of programming assessments at introductory levels, whether or not inherent in the subject itself, presents a significant and possibly unfair barrier to student success.*

Chapter 3

Students Worldwide Do Not Learn to Program

The previous chapter made the theoretical observation that programming courses have demanding goals. This is borne out by concrete evidence: poor results in introductory programming education have been widely reported. Section 3.1 below reviews research on learning outcomes, which indicates that many students are not learning to write programs in CS1. Section 3.2 considers the relationships between code-reading skill and code-writing skill, but sadly, it turns out in Section 3.3 that many students do not even learn to read program code reliably in CS1. Finally in Section 3.4, I review research on the various specific problems that novice programmers have with understanding fundamental programming concepts.

3.1 Many students do not learn to write working programs

> *Few teachers of programming in higher education would claim that all their students reach a reasonable standard of competence by graduation. Indeed, most would confess that an alarmingly large proportion of graduates are unable to 'program' in any meaningful sense.* (Carter and Jenkins, 1999)

This is not recent news. According to the review of CER literature from the 1980s by Robins et al. (2003), "an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course". Linn and Dalbey (1985) report that most students struggled to get past learning language features and never got to learning about higher-order skills of program planning and general problem-solving strategies for programming. Kurland et al. (1986) concluded that after two years of programming instruction, many high-school students had only a rudimentary understanding of programming. Guzdial reviews the work of Soloway and others:

> *One of the first efforts to measure performance in CS1 was in a series of studies by Elliot Soloway and his colleagues at Yale University. They regularly used the same problem, called "The Rainfall Problem": Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average. In one study, only 14% of students in Yale's CS1 could solve this problem correctly. The Rainfall Problem has been used under test conditions and as a take-home programming assignment, and is typically graded so that syntax errors don't count, though adding a negative value or 99999 into the total is an automatic zero. Every study that I've seen (the latest in 2009) that has used the Rainfall Problem has found similar dismal performance, on a problem that seems amazingly simple.* (Guzdial, 2011, see also Soloway et al., 1982; Venables et al., 2009)

A common topic of conversation among computing education researchers is the 'Bactrian' grade distribution of many introductory programming courses: students either fail miserably or pass with flying colors, with few 'just doing okay' (Dehnadi and Bornat, 2006). Many explanations have been offered and many studies have been conducted, but the phenomenon remains unexplained (see, e.g., Bornat et al., 2008; Robins, 2010).

Multi-institutional studies

In the past decade, several international working groups have looked into the skill levels of students at the end of CS1 courses, or, in some cases, at the end of a degree program (McCracken et al., 2001; Lister et al., 2004; Eckerdal et al., 2006b). These oft-cited studies have been influential as they produced concrete evidence of the mismatch between the goals of programming education and the actual skills gained.

In 2001, a multi-institutional, multi-national working group chaired by Michael McCracken gave a set of program-writing problems of varying difficulty to students completing CS1 or CS2 in several countries. The students only got an average score of approximately 23 out of 110 points. The authors state that their "first and most significant result was that the students did much more poorly than we expected" and that "the disappointing results suggest that many students do not know how to program at the conclusion of their introductory courses" (McCracken et al., 2001).

Another international working group study explored program design skills. When Eckerdal et al. (2006b) analyzed the designs produced by students, they found "poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design". They conclude that "the majority of graduating students cannot design a software system". A recent follow-up study by Loftus et al. (2011) produced similar results.

It is clear by now that learning to write programs is a challenging goal. Teachers aware of this have tried to find ways of easing the burden of students as they gradually develop code-writing ability. The question then becomes: what are the relationships between the various goals of programming education? Or more specifically: what other skills does the skill of writing code build on? There is some limited evidence of dependencies between programming skills, so that learning certain skills builds on learning others first.

3.2 Code-writing skill is (loosely?) related to code-reading skill

> *The ability to write program code is what we aim to teach, so anything else that we can discover about students' acquisition of skills must ultimately be considered in the light of their ability to write code.* (Lister et al., 2009a)

Many programming teachers find it intuitive, even self-evident, that one must learn to read code before one can write code, just as one learns to read their first natural language before learning to write in it. Similarly, the ability to trace a program's execution steps would seem to be a prerequisite both for explaining what given code accomplishes and for writing code. However, students of programming tend to give examples and reading tasks little attention compared to writing; some also say right out that writing code is easier than reading (Simon et al., 2009).

Does learning to explain what a piece of code does precede learning to write code? Can people learn to write code successfully without being able to trace its execution (and code of what kind)? What comes before the ability to explain code? More generally, is there evidence of a general learning path of programming skills?

From taxonomy to learning path?

Bloom's taxonomy (Section 2.1) is not of much assistance in the search for a general learning path. Even if we assume that the cognitive categories form a hierarchy of increasingly complex learning objectives and assessments, there is no evidence in the general case that learning a higher-ranking skill requires the lower-ranking skills to be learned first. It has been argued that Bloom's taxonomy does not match the path(s) of skill progression that learners take, either in general or for programming in particular (e.g., Fuller et al., 2007; Anderson et al., 2001; Eckerdal et al., 2007; Biggs and Tang, 2007).

Other frameworks may match the learning process better. A substantial body of empirical research that seeks to discover how programmers' skills develop has recently been produced by the BRACElet project (for an overview, see Clear et al., 2011). This empirical work builds on SOLO (Section 2.2), a taxonomy that was designed to reflect stages of learning.

BRACElet: some evidence of skill dependencies

In terms of one interpretation of SOLO (Section 2.2), code-tracing skills (that is, the ability to step through a program's execution) rank below code-explaining skills (that is, the ability to determine and state the overall purpose of a piece of code), as the former requires multistructural learning outcomes while the latter requires relational ones. If SOLO levels represent a learning path, students would need to learn to trace code of a particular kind and complexity before they learn to explain code of a similar kind and complexity. The BRACElet members hypothesized that learners generally first learn to trace programs, then to explain them, and finally to write them.

A number of BRACElet publications have produced empirical evidence that links the skills of tracing, explaining, and writing code. Students who write programs successfully tend to be able to produce correct overall explanations of what a given piece of code does (Whalley et al., 2006). Students' abilities to explain and write correlate positively (Sheard et al., 2008). Students who cannot trace code are usually also unable to explain what a given piece of code does (Philpott et al., 2007). Summarizing what given code does appears to be an intermediate-level skill, which programming experts use naturally in lieu of line-by-line traces of programs, but which many novices struggle with (Lister et al., 2006b).

Lopez et al. (2008) report that performance level in a code-tracing task accounts for some of the variation in code-explaining performance. Further, they found that code-explaining ability alone, or code-tracing ability alone, appears to account for only a little of the variation in code-writing skills. However, explaining and tracing abilities together account for a substantial amount of the variation in writing ability. They conclude that if one posits that a causal model exists between the skills, then their findings support a hierarchy where tracing is lower than explaining, which is again lower than writing. The later results of Lister et al. (2009b) and Venables et al. (2009) are consistent with the analysis of Lopez et al. In sum, the BRACElet research points "to the possibility of a hierarchy of programming related tasks where knowledge of programming constructs forms the bottom of the hierarchy, with 'explain in English', Parson's problems, and the tracing of iterative code forming one or more intermediate levels in the hierarchy" (Clear et al., 2011).

Simon et al. (2009) followed up on these studies. They found that – contrary to expectations – no meaningful relationships could be established when comparing the marks of a writing task and a comparable reading task (a 'Parson's problem', which requires the sorting of lines of code, and has been shown to assess similar skills to traditional code-writing questions; see Denny et al., 2008). This result may be explained by: 1) the lack of established criteria for comparing the complexity of two comparable programs from a learning point of view; 2) the differences between marking code-reading problems on the one hand and code-writing problems on the other (Simon et al., 2009), and 3) the ambiguities of code-explaining questions in general (Simon, 2009).

Other researchers have found some evidence to support the claim that novices can produce code based on familiar templates even without being good at tracing it (Anderson et al., 1989; Thomas et al., 2004). It can be questioned, however, whether such novices can reliably produce bug-free code and whether they can fix all the bugs in the code they produce without successfully tracing the programs.

As the BRACElet authors themselves have noted, the results from their line of research are not straightforward to interpret, and the matter of a learning hierarchy of programming skills remains unsolved. Furthermore, research has so far has focused on simple tracing, code-explaining, and writing tasks, and little has been shown about the relationship of these skills to other programming skills such as debugging, program design, or dealing with larger amounts of code. Nevertheless, it is easy to agree at least with the cautious conclusion of Venables et al.:

> *In arguing for a hierarchy of programming skills, we merely argue that some minimal competence at tracing and explaining precedes some minimal competence at systematically writing code. Any novice who cannot trace and/or explain code can only thrash around, making desperate and ill-considered changes to their code – a student behavior many computing educators have reported observing.* (2009, p. 128)

3.3 But many students do not learn to read code, either

In the light of the previous chapter, it is not very surprising that learners fail to learn to write and design programs. After all, these are cognitively complex skills that require relational understandings of structurally complex content. What about presumably less complex programming skills, such as tracing and explaining? Even though a strict learning hierarchy may not exist, the skill of program writing is partially dependent on these less complex skills. Are introductory programming courses succeeding in teaching students to read code?

Following up on the McCracken investigation described in Section 3.1, Lister et al. (2004) measured students' ability to trace through a given program's execution. They gave a multiple-choice questionnaire to students in a number of educational institutions around the world. The questions required the students, who were near the end of CS1, to predict the values of variables at given points of execution and to complete short programs by inserting a line of code chosen from several given alternatives. A quantitative analysis of the multiple-choice questions was complemented by student interviews and an analysis of the 'doodles' students made while tracing. Lister et al. found that many students were unable to trace. While there was obviously some variation in students' ability between institutions, the results were disappointing across the board.

Other studies have produced similar results. For instance, an earlier multi-institutional study by Sleeman et al. (1986) analyzed code-driven interviews to conclude that "at least half of the students could *not* trace through programs systematically" upon request and instead "often decided what the program would do on the basis of a few key statements". Adelson and Soloway (1985) found that novices were unable to mentally trace interactions within the system they were themselves designing. Kaczmarczyk et al. (2010) report an inability to "trace code linearly" as a major theme of novice difficulties. The analyses of quiz questions by Corney et al. (2011), Simon (2011), Teague et al. (2012), and Murphy et al. (2012) indicate that many students fail to understand statement sequencing to the extent that they cannot grasp a simple three-line swap of variable values. In one study, the problem existed even among students taking a third programming course (Simon, 2011).

It is clear from these results that students' foundational programming skills commonly do not develop as well as teachers expect. These results are discouraging, but cannot be ignored; of particular significance is the fact that many of the findings have been produced by international, multi-institutional studies which demonstrate that the problem is not local to a particular university, country, or type of institution.

3.4 What is more, many students understand fundamental programming concepts poorly

So far in this chapter, we have focused on skills. What about conceptual understanding? We gain another perspective on the challenge of introductory programming education by looking at what is known about novices' conceptions of programming concepts.

In some disciplines, concepts and phenomena are largely negotiable and up for interpretation. Students may be encouraged to interpret things in a personal way and to develop alternative conceptual frameworks. Certainly, many computing concepts are like this, too. However, computing also features many concepts that are precisely defined and implemented within technical systems. Students are expected to reach particular ways of understanding what the assignment statement in Java does, of what an object is, and of how a given C program executes. Sometimes a novice programmer 'doesn't get' a concept, or 'gets it wrong' in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs. Unfortunately, misconceptions of even the most fundamental programming concepts, which are trivial to experts, are commonplace among novices and challenging to overcome. Recent studies measuring students' conceptual knowledge suggest that introductory programming courses are not particularly successful in teaching students about fundamental concepts, and that the problems are not limited to a single institution nor caused by the use of a particular programming language (Elliott Tew, 2010; Kunkle, 2010).

Over the past few decades, many researchers have catalogued ways in which programming students

struggle with fundamental concepts, and the kinds of incomplete and incorrect understandings that students have exhibited. Variables, assignment, references and pointers, classes, objects, constructors and recursion are among the CS1 concepts most commonly reported as problematic. Many students appear to have problematic understandings about the capabilities of computers and programs in general.

What follows is a review of research on the misconceptions and limited understandings that novice programmers have been found to have about fundamental programming concepts. Some examples of misconceptions appear below; there is a more comprehensive list in Appendix A. The reader can find a longer review of the earlier work on misconceptions within CER in a book chapter by Clancy (2004).

Research on programming misconceptions

Bayman and Mayer (1983) studied beginners' interpretations of statements in the BASIC language by asking students to write plain English explanations of programs. They list a number of misconceptions about BASIC semantics, e.g., `LET` statements are understood as storing equations instead of assigning to a variable. Around the same time, Soloway, Bonar, and their colleagues also explored novice misconceptions and bugs, and discussed how they may be caused by knowledge from outside of programming, particularly by analogies with the everyday semantics of natural language (e.g., Soloway et al., 1982; Bonar and Soloway, 1985; Soloway et al., 1983).

Samurçay (1989) studied the answers that programming beginners gave to three program completion tasks, and reported that variable initialization in particular was difficult for students to grasp. Putnam et al. (1986) and Sleeman et al. (1986) analyzed students' answers to code comprehension tests and subsequent interviews. They listed numerous errors – surface and deep – that students make with variables, assignment, print statements, and control flow. Du Boulay (1986) likewise listed a number of novice misconceptions with variables, assignment, and other fundamental programming concepts.

Pea (1986) listed a number of common novice bugs and suggested that they are rooted in a 'superbug', the assumption that there is a hidden, intelligent mind within the computer that helps the programmer to achieve their goals.

Fleury (1991) and Madison and Gifford (1997) interviewed and observed students to discover various conceptions of parameter passing. Their results suggest that even students who are sometimes capable of producing working code with parameters may misunderstand the concepts involved in different ways.

Recursion has been the focus of several studies. Kahney (1983) discovered that students have various flawed models of recursion, such as the "looping model", in which recursion is understood to be much like iteration. Kahney's work has since been elaborated on by Bhuiyan et al. (1990), George (2000a,c), and Götschi et al. (2003). Recursion is also one of the phenomena investigated by Booth (1992) in her phenomenographic work on learning to program. Booth identified three different ways of experiencing recursion: as a programming construct, as a means for repetition, and as a self-reference; students are not always able to grasp all of these aspects.

Recent themes: OOP and Java

Since the '90s, interest in CER has shifted from procedural programming towards object-oriented programming. Several studies have reported ways in which students misunderstand object-oriented concepts and features of OO languages. Holland et al. (1997) noted several misconceptions students have about objects. For instance, students sometimes conflate the concepts of object and class, and may confuse an instance variable called `name` with object identity. More novice misconceptions about OOP were reported by Détienne (1997) as part of her review of the cognitive consequences of the object-oriented approach to teaching programming.

Fleury (2000) reported that students form their own, unnecessarily strict rules of what happens in programs and what works in Java programming. For instance, some of the students she studied thought that the dot operator could only be applied to methods and that the only purpose of a constructor was to initialize instance variables.

Hristova et al. (2003) listed a number of common errors students make when programming in Java. Many of these are on a superficial syntactic level (e.g., confusing one operator with another) but some suggest deeper-lying misconceptions. Ragonis and Ben-Ari (2005a) reported the results of a wide-scope,

long-term, action research study of high-school students learning object-oriented programming. Their study uncovered an impressive array of misconceptions and other difficulties students have with object-oriented concepts, the Java language, and the BlueJ programming environment.

Teif and Hazzan (2006) observed students of introductory programming in two high-school courses and discuss students' conceptual confusion regarding classes and objects. For instance, students may incorrectly think that the relationship between a class and its instances is partonomic, i.e., that objects are parts of a class. Eckerdal and Thuné (2005) also studied the conceptions that students have of these fundamental object-oriented concepts. Their results highlight the fact that not all students learn to appreciate objects and classes as dynamic execution-time entities or as modeling tools that represent aspects of a problem domain.

Vainio (2006) used interviews to elicit students' mental models of programming concepts. Among his results are a number of misconceptions about fundamental concepts, e.g., the idea that the type of a value can change on the fly (in Java).

Several recent, complementary studies affirmed the existence of a number of incorrect understandings of assignment, variables, and the relationships between objects and variables. Ma (2007) gave a large number of volunteer CS1 students a test with open-ended and multiple-choice questions about assignment in Java. He analyzed the results both qualitatively and quantitatively to understand students' mental models of assignment and reference semantics. I myself explored students' understandings of storing objects and of Java variables through phenomenographic analyses of interviews (Sorva, 2007, 2008). Doukakis et al. (2007) combined findings from the literature and anecdotal evidence to list introductory programming misconceptions, which, the authors argue, arise as a result of students' prior knowledge of mathematics.

Sajaniemi et al. (2008) elicited the mental models that novice programmers have of program state by having CS1 students draw and write about how they perceived given Java programs' state at a specific stage of execution. They discovered numerous misconceptions relating to parameter passing and object-oriented concepts.

As part of a project to develop a concept inventory for CS1, Kaczmarczyk et al. (2010) used interviews to identify student misconceptions about concepts and grouped the misconceptions thematically. Four themes were identified: the relationship between language elements and memory, while loops, the object concept, and code-tracing ability.

Viability

An understanding – even a misconception – is usually not universally useless. It may be viable for a particular purpose but non-viable in the general case. People may be entirely satisfied with their misconceived notions, even for a considerable amount of time. Nevertheless, the kinds of understandings of fundamentals reviewed here and in Appendix A are worrisome because they will cause problems for the novice programmer usually sooner rather than later. Those non-viable understandings need addressing; failure to do so is a failure of programming education.

The literature is effectively unanimous. Introductory programming education is struggling to cope with the challenging task of teaching beginners to create programs. Even teaching students to read programs is a challenge, as is helping students form viable understandings of fundamental programming concepts.

Part II

Learning Introductory Programming

Introduction to Part II

What is involved in learning to program? Under what circumstances do students succeed? What are the major obstacles to learning programming?

The following chapters delve more deeply into what learners go through as they study programming. I review what several learning theories have to say about learning in general, and what computing education research has to say about learning to program in particular. I also consider, in fairly generic terms, the pedagogical recommendations that arise from the theories. This review lays the foundations for the discussion of approaches to teaching CS1 in Part III.

Part II consists of seven chapters.

Chapter 4 focuses around schema theory, which explains how people learn to solve complex problems despite the limitations of the human cognitive architecture. I stay with cognitive psychology in Chapter 5, which deals with the mental models that people form of the systems they interact with; this leads to a discussion of how programming requires a viable mental model of how the computer executes programs. In Chapter 6, I turn to the family of educational theories known as constructivism, whose emphasis on learner-driven pedagogy has grown to be influential in many fields of education, including computing education. Chapter 7 is concerned with the phenomenographic tradition of empirical research on education. In the phenomenographic view, the most important form of learning involves becoming able to experience particular content – such as computer programs – in new ways.

There is, of course, a vast number of other traditions and theories within psychology, education, and CER; I cannot possibly discuss more than a few. The theories and research traditions that I have selected for Part II are particularly relevant because they each help us understand learning from a different perspective. Each of them is also relatively influential in programming education research: the cognitive perspective is well established, constructivism has recently greatly grown in influence, and a body of phenomenographic work is also emerging in the CER literature.

Although there is common ground, the different theories and theorists are not in full agreement with each other. Especially since I draw on multiple perspectives in a single thesis, it is important to consider the merits and weaknesses and compatibilities and incompatibilities of each point of view. Chapters 5, 6, and 7 each conclude with a section in which I review some of the main criticisms that have been leveled at cognitivism, constructivism, and phenomenography, respectively. The short interlude that is Chapter 8 compares and contrasts what these perspectives have to say about learning and programming.

Chapter 9 rounds off Part II with a brief discussion of the theory of threshold concepts, an emerging framework which seeks to explain why learners sometimes get 'stuck' at particular points of a curriculum, and which has eclectically drawn from many sources, including the ones discussed in the preceding chapters.

Chapter 4

Psychologists Say: We Form Schemas to Process Complex Information

> *Contrary to popular belief, the brain is not designed for thinking. It's designed to save you from having to think, because the brain is not actually very good at thinking.* (Willingham, 2009, p. 3)

A massive, multi-threaded literature on learning has emerged from cognitive psychology. Theories of working memory, schemas, cognitive load, and mental models – among others – have contributed to our understanding of what it means and what it takes to learn. There is also a body of research on the psychology of programming that has applied general psychology to computer programming and come up with theoretical models that explain how people program and how they learn to program.

This chapter is the first of two chapters on cognitive psychology. Section 4.1 provides a short introduction to the idea of mental representation and the multi-store model of human memory, two cornerstones of modern psychology. Section 4.2 is an introduction to schema theory, followed by a discussion in Section 4.3 of the role of mental schemas in problem solving and in the growth of expertise. In Section 4.4 I turn from general schema theory to the psychology of programming: I review what is known about how people apply and construct schemas as they write programs. Section 4.5 introduces cognitive load theory, an influential 'spin-off' of schema theory with important and concrete implications for instructional design. Finally, in Section 4.6, I discuss existing research on program comprehension models, that is, theoretical models of how people make sense of computer programs. We will get to the theory of mental models in the next chapter.

4.1 Cognitive psychology investigates mental structures

A large body of psychological research deals with the ways in which knowledge is represented in the human cognitive system and the way people process these *mental representations*. Our use of mental representations is constrained by how human memory works – another topic fundamental to much of cognitive psychology. A sound theory of mental representation helps explain why we can function effectively despite the significant limitations of the architecture of our memory.

4.1.1 Inside minds, there are mental representations

Markman (1999, pp. 5-11) defines a *representation* as something that: 1) is about and captures aspects of a *represented world*; 2) exists in a *representing world* that is an abstraction of the represented world; 3) relies on rules that map elements in the represented world to elements in the representing world, and 4) is accompanied by *processes* that allow people or intelligent systems to make sense of the representation. The height of mercury in a thermometer, for instance, is a (non-mental) representation of a certain aspect of our physical world in the more abstract representing world of the thermometer. Representing rules map the possible heights of the mercury to different temperatures, and humans use an interpretative process associated with the representation to read the temperature. In Markman's definition, the difference

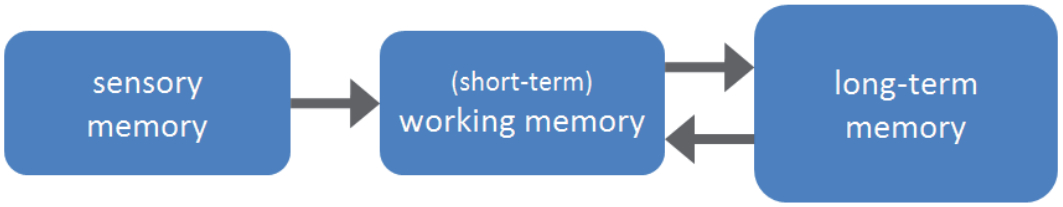


Figure 4.1: A commonly used basic architecture of human memory.

between a cognitive (mental) representation and other representations is a disciplinary one: a mental representation needs to be explained primarily by psychology rather than, say, the physical sciences.

There is a vast number of theories of mental representation. Some of the theories complement each other, some are in competition. The mental representations postulated by the various theories vary in terms of the duration of their existence, their discreteness, and their degrees of genericity and abstraction (Markman, 1999, pp. 14-16). The theories also vary in terms of which aspects of human behavior they seek to explain. For instance, schema theory (this chapter) is primarily concerned with how people gradually become able to deal with increasingly complex situations, while mental model theory (Chapter 5) primarily characterizes the human ability to deal with novel situations and causal systems. The program comprehension models of Section 4.6 are examples of programming-specific theories of mental representation.

4.1.2 Working memory can only hold a handful of items at a time

Since the 1960s, educational psychology has been greatly influenced by the distinction between short-term and long-term memory, two parts of the so-called multi-store model that was seminally formulated by Atkinson and Shiffrin (1968) and later extended in various ways by others. Figure 4.1 shows a typical diagram of the multi-store model, which consists of three parts.

Sensory memory is transient and stores sensory input extremely briefly, for less than a second. Its contents serve as input to the other memory systems.

Long-term memory is a memory system that is capable of storing large amounts of information for very long times. Its capacity is vast. However, we are oblivious to most of what is in our long-term memory most of the time. In order to use it, we must retrieve it for processing.

Such processing happens in *working memory*, an intermediate storage that is very limited in duration and capacity. People use working memory as they actively manipulate information that has just come in through the senses or that has been retrieved from long-term memory.[1] The most well known theory of working memory is Baddeley's. In his model, working memory consists of a *central executive* system which controls three other subsystems: a *visuospatial sketch-pad*, where we 'see' what we are currently thinking about visually, a *phonological loop*, where we 'hear' what we are currently articulating or hearing, and an *episodic buffer* that is capable of short-term integration of information from multiple sources (see, e.g., Baddeley and Hitch, 1974; Baddeley, 2000). Without rehearsal, information in working memory is lost in a matter of seconds (Peterson and Peterson, 1959).

George Miller's famous paper estimated the number of items that working memory can simultaneously hold to be "the magical number seven plus or minus two" (Miller, 1956). Later estimates have generally been even lower. Some theories do not count items; for instance, the concept of working memory in the influential ACT-R model of cognition is based on the "degrees of activation" of items in long-term memory – it is the total degree of simultaneous activation that is strictly limited (Anderson et al., 1996). It has also been suggested that the auditory part of working memory is temporally constrained not by a magic number but a "magic spell" (Schweickert and Boruff, 1986). Although the theories differ in their

[1] Working memory is often equated with *short-term memory*. Some theories distinguish between the two concepts, however. Ericsson and Kintsch (1995), for instance, argue for the existence of *long-term working memory*, a domain-specific extension of short-term memory within long-term memory that is created through extensive practice. This distinction between short-term memory and working memory is unimportant for my present purposes. Where I write "working memory", one may read "short-term working memory" instead.

details, there is a general agreement that working memory capacity is very limited, and that this limitation is a key feature of the human cognitive system.

4.1.3 Chunking and automation help overcome the limitations of working memory

How can we accomplish anything at all sophisticated with such a limited working memory? The answer, according to cognitive psychology, lies in the mechanisms of *chunking* information (Miller, 1956) and *automation* of processing (Schneider and Shiffrin, 1977).

To remember a phone number or date, we do not memorize a sequence of individual integers such as 0-4-0-0-5-1-5-3-8-8 or 0-6-0-3-2-0-1-0, but group the integers so that they form bigger *chunks*: 0400-515-388 or 06-03-2010. Chunks often, but not always, carry meaning on their own (e.g., month, operator code). A chunk, although composite, can be treated as a single item by working memory, allowing us to process larger amounts of information simultaneously. As we become increasingly familiar with information, we process it increasingly automatically, without paying attention to its components and without having to use our working memory. Tuovinen (2000) uses fluent readers as an example: "they do not try to read out individual letters, but process larger groups, words or groups of words, without attending to individual letters or even words separately."

What does this mean for learning? We learn something when we store it in long-term memory. Working memory is a bottleneck that limits our access to long-term memory. Cognitive science suggests that increasing one's ability to deal with information in large chunks and ever more automatically is key to learning and the growth of expertise. The following sections elaborate on this claim.

4.2 We store our generic knowledge as schemas

> *Schemas represent knowledge as stable patterns of relationships between elements describing some classes of structures that are abstracted from specific instances and used to categorize such instances.* (Kalyuga, 2010, p. 48)

A *schema* is a mental structure that contains generic conceptual knowledge (see, e.g., Rumelhart and Ortony, 1977; Rumelhart and Norman, 1978; Rumelhart, 1980; Anderson, 1977; Kalyuga, 2010). People make use of schemas constantly, both as they reason about everyday objects and situations, and as they solve problems. A person can have a schema of what a house is, of what going to a restaurant typically involves, or of how to write a program that computes an average of given numbers. A schema of houses, for example, may include the knowledge that a house is a type of building, that it consists of rooms, has walls and windows, is typically made of wood, brick, or stone, probably has a rectilinear shape, normally functions as a human dwelling, and is likely to be between 100 and 10,000 square feet in size (example from Anderson, 2009, pp. 134–135).

A schema contains knowledge of the stereotypical properties and parts of the concept, process, or situation it represents. The properties of a schema may involve other schemas, creating hierarchical structures or networks of schemas; the concept of house is linked to the concepts of walls and windows, for instance. Schemas may link to themselves as well, forming recursive structures (see, e.g., Rumelhart and Ortony, 1977). Schemas reside in long-term memory; the human mind is capable of storing countless complex schemas.

A generic schema allows us to make inferences about specific instances. For example, if someone speaks of their own house, we can – and do – fairly safely assume certain things about it, such as its shape (but we may be wrong, too). A schema "determines expectancies, organizes encoding, and systematically distorts retrieval [of knowledge from memory] in the direction of internal consistency." (Hintzman, 1986, p. 424). Schemas do allow for unusual properties in specific instances. If we see a house made of glass, we can still easily consider it a house, albeit untypical. Just how untypical an instance is allowed to be to still be considered a member of a schema-based category depends on the schema and the individual, and possibly even the time of day, as people do not always categorize instances consistently (see, e.g., Anderson, 2009, p. 136–138).

Schemas are created, extended, and modified through experience. Schema theorists vary on the details, but they all make the same central claim: previously existing schemas in long-term memory

play a decisive role in how and where new experiences are integrated into people's mental representations. Rumelhart and Norman (1978) identify three modes of learning. *Accretion* is the common, straightforward accumulation of new information into existing schemas as more and more instances of a familiar concept are encountered. *Tuning* means the continual minor adjustment of one's existing schemas by constraining and generalizing the generic conceptual categories that they represent and adjusting the typicality of the values of schema properties. Finally, *restructuring* is a more dramatic change in mental representation that results in novel conceptualizations. It is triggered by unfamiliar experiences and involves the creation of new schemas, the reorganization, modification, or deletion of old schemas, and possibly abstraction from known concepts. People tend to resist such radical restructuring, and dramatic changes in schemas may take great time and effort. In all three forms of learning, the schemas that we already have determine how we deal with new experiences.

Having a schema does not mean you 'got it right'. New information is encoded – via accretion, tuning and restructuring – in terms of existing domain-specific schemas, but not necessarily the ones the teacher intended. Misconceptions (such as those discussed in Section 3.4; see also Appendix A) form when the learner integrates new information into the wrong existing schema or uses a non-viable analogy to motivate the creation of a new schema.

Broad vs. narrow definitions of schemas

The development of schema theory and the differing definitions given for schemas in the literature have been discussed by Brewer (2002), among others; my short review below draws primarily on Brewer.

The origin of schema theory is typically attributed to Bartlett's 1932 book *Remembering*, which predates modern cognitive psychology by a couple of decades. Bartlett found that people filled in stories with imaginary details they assumed to be true on the basis of the mental frameworks they used for understanding and remembering information. Later, Ausubel's (1968) subsumption theory of learning, which involves hierarchical memory structures and emphasizes the role of prior knowledge, foreshadowed the development of schema theory, as did the work of Jean Piaget. In the 1970s, Bartlett's ideas were revitalized by Minsky (1974), who sought to model humans' cognitive structures in computer memory as a part of the artificial intelligence movement. Schema theory was then established as a bona fide theory of learning within the research communities of psychology and education in seminal papers by Rumelhart (e.g., 1980) and Anderson (e.g., 1977). Around this time, it was noticed that schema theory explained the results of many previous experiments, and it led to a proliferation of empirical studies. Since the turn of the '80s, schema theory has made a substantial impact on educational psychology and CER. Over the past two decades, cognitive load theory (Section 4.5 below) has emerged as an influential offshoot of schema theory.

Perhaps because varied groups of people have been interested in the concept over a number of decades of early development, the word "schema" is not consistently used in the literature. There is a myriad of ways in which schemas are described, but uses of the term appear to fall roughly in one of two categories. The first corresponds to the way I have used the word above. In this usage, a schema is a mental representation for 'old', generic conceptual knowledge. The knowledge is old in the sense that it is based on previous experience and stored in long-term memory (rather than new in the sense of being currently experienced and constructed in working memory), and generic in the sense that it deals with conceptual categories rather than specific instances or contexts. The other, broader way to use "schema" is to umbrella all kinds of knowledge structures with it. For instance, Derry (1996) describes schemas as "virtually any memory structures", examples including generic schemas (as per the narrower definition above), situation-specific mental models, and phenomenological primitives (see Section 6.5).

Some authors have addressed this confusing terminological issue explicitly (e.g., Brewer, 1987, 2002; Choi and Sato, 2006). Brewer notes that even the original authors of seminal schema theory papers were inconsistent in their use of the term. He suggests that the narrow meaning is preferable, since the broad interpretation of "schema" is redundant and problematic with respect to the explanatory power of schema theory. I tend to agree with Brewer, and will continue throughout this thesis to use the word "schema" in the narrower sense to mean structures for old generic knowledge.

4.3 Schemas keep the complexity of problem solving in check

> *The discovery that the major difference between people who differed in ability was in terms of what they held in long-term memory changed our view of cognition.* [...] *Long-term memory was not just used by humans to reminisce about the past but, rather, was a central component of problem solving and thought.* (Sweller, 2010a, p. 32)

4.3.1 Scripts and problem-solving schemas tell us what to do

The literature discusses various more specific types of schemas. An often-cited construct is the *script* or *event schema* formulated by Schank and Abelson (1977). Scripts are schemas that encode knowledge of recurring events and their stereotypical stages and other properties, and allow us to act comfortably and efficiently in familiar situations. The event of a visit to a restaurant, which consists of typical stages like ordering, eating, and paying, is a classic example. Similarly, a *problem-solving schema* tells us what to do to accomplish a goal in a particular kind of situation. Sweller and Chandler provide a good example:

> *Someone who is competent at algebra will have a schema for multiplying out a denominator. The schema will tell that person which of the infinite variety of algebraic equations is amenable to multiplying out a denominator and the procedure for doing so. When faced with a problem such as* a/b = c*, solve for* a*, we can immediately solve such a problem, despite the many forms in which it could be presented, because our schema for this type of algebra problem informs us, for example, that the solution requires multiplying out the denominator on the left-hand side, irrespective of the complexity of the term on the right-hand side. Schemas, stored in long-term memory, permit us to ignore the variety that would otherwise overwhelm our working memory.* (Sweller and Chandler, 1994, p. 187)

Schemas allow us to draw on our prior experience and general knowledge so as not to be overwhelmed by the details of what we are currently experiencing. When we see an unfamiliar person – a specific instance of a familiar, automated schema – we can easily tell that the person is a human and act accordingly, without having to store or process every tiny detail of the person that our senses barrage us with. Schemas are crucial to our success in whatever we do as they "allow us to carry out everyday activities with minimum effort and to capitalize on the regularities of events and situations" (Preece et al., 1994, p. 128).

When faced with a complex problem to solve, we need to process various aspects of the problem and its solution in working memory simultaneously. A problem-solving schema such as the algebra schema described above allows us to ignore the specific details of the problem and recognize a more general pattern for which we have a schema. When a schema is available, it allows the entire problem or a part of the problem to be dealt with as a single chunk (see Section 4.1 above). In relation to one of their experimental studies, Sweller and Chandler (1994) discuss the solving of a simple problem: marking a point with given coordinates in a two-dimensional graphical coordinate system. They argue that for a complete novice, solving this problem requires an understanding of roughly seven distinct items, e.g., the fact that x in *P(x, y)* refers to a location x on the x-axis. Where the complete beginner may be overwhelmed, someone who has experience with such problems can incorporate the entire problem into a single chunk that does not burden working memory greatly – or at all, if automated through practice.

4.3.2 Schemas are a key ingredient of expertise

Educational psychologists have long sought to understand the nature and development of expertise. On the one hand, the goal has been to characterize the mental representations that experts have and the ways in which experts put their knowledge to use when solving problems. On the other hand, researchers have investigated the mental representations and associated processes of novices, and contrasted them with those of experts. Through such comparisons, it is argued, we may better understand the changes that take place as we learn, and thereby improve teaching and learning.

An easy explanation for the superior performance of experts in problem solving would be in their superior 'cognitive hardware' such as a larger short-term working memory or superior intelligence. Early

cognitive psychology also sought to explain expertise through general, domain-independent problem-solving skills and thinking strategies. Undoubtedly, talent does play a part in the development of expertise, and there is some transfer of expertise between related domains. However, many present-day cognitive scientists argue that the main factor that sets experts apart from non-experts is not their 'hardware' or any generic strategy that they possess, or their innate aptitude for a domain, but a readily accessible and usable domain-specific knowledge base in the form of schemas, acquired through lengthy practice.

The results of Chi et al. (1982; Glaser and Chi, 1988), who studied experts and novices solving physics problems, provide evidence of the key role that schemas play in the development of expertise. Chi et al. found that both experts and novices use schemas of relevant types of physical objects (inclined planes, for instance), but only experts use schemas which deal with more general principles such as the conservation of energy, and which subsume object schemas. Correspondingly, novices classified problems (that is, chose which schemas to activate) on the basis of the surface features of problems while experts were more concerned with the underlying patterns and principles. As schemas form organized hierarchies, experts could search for the required knowledge efficiently and quickly.

According to the psychological literature reviewed by Glaser and Chi (1988) and Winslow (1996), the growth of expertise is a process that involves the formation of many mental representations, deep and interlinked knowledge hierarchies, sophisticated strategies, and a rich arsenal of problem-solving patterns. Experts perceive – within their own domain of expertise – large, meaningful patterns that permit fast and error-free problem solving. They analyze problems in terms of principles rather than surface structures and tend to spend a lot of time analyzing problems qualitatively. In contrast, novices' untransferable 'fragile knowledge' (Perkins and Martin, 1986), lack of adequate mental representations, and tendency to use generic and inefficient problem-solving strategies have been widely observed. These characteristics of novices and experts have been documented in many disciplines, from chess to sports to programming.

The capacity of experts' working memory is as limited as that of novices, but experts compensate for this through efficient schema-based chunking. As experience grows, one forms schemas at ever higher levels of abstraction, which allows representing ever larger parts of a problem or its solution as individual chunks. Experts' high degree of automation further alleviates the load on working memory.

A corollary of the schema theory view of expertise is that since expertise is tied to the extent and quality of one's knowledge base, expertise in one domain is not readily transferable to another. A number of studies corroborate this claim (see, e.g., Glaser and Chi, 1988; Anderson, 2009, pp. 263–265).

4.3.3 An introductory course starts the novice on a long road of schema-building

Degree programs do not generally create fully-fledged experts. Dreyfus and Dreyfus (2000) identify five stages in the development of expertise: 1) a *novice* learns context-independent facts and rules; 2) an *advanced beginner* starts to recognize more abstract patterns in situations; 3) a *competent* problem-solver is capable of considering wholes and consciously choosing plans for achieving goals; 4) the level of *proficiency* is characterized by the automation of planning, which is no longer completely conscious, and 5) an *expert* has a mature and practiced understanding that allows them to 'see' what to do. Reaching expertise is a long process; cognitive psychologists speak of the "ten-year rule" in reference to the roughly ten years, or 10,000 hours, of deliberate practice that it takes to become an expert in a domain (see Willingham, 2009; Ericsson and Kintsch, 1995; Palumbo, 1990, and references therein). The goal of an undergraduate program, then, is not (or should not be) to create experts but to help students take some steps towards expertise and to prepare them to complete the journey on their own. As Winslow (1996) says, "most of us would probably settle for a graduate who ranks between competent and proficient".

Does it even make sense to speak of expertise in the context of introductory courses, the focus of my present work? I believe it does. 'Full expert status' may not be a reasonable goal for entire degree programs, and certainly is not one for introductory programming courses. Nevertheless, studies of expertise do give us a sense of the direction in which novice learners should progress. Schema theory suggests that the path to expertise is a path of knowledge-building. Novices should be helped to form fundamental schemas that can serve as the building blocks of more complex ones, and gain practice in their use so that they do not need to worry about the smallest details while solving increasingly complex problems.

Now let us turn to programming.

```
count = 0
sum = 0

number = float(raw_input('Enter a number:'))
while number < 99999:
    sum = sum + number
    count = count + 1
    number = float(raw_input('Enter a number:'))

if count > 0:
    average = sum / count
    print average
else:
    print 'No numbers input: average undefined.'
```

Figure 4.2: This program, which determines the average of given numbers, features several programming plans (adapted from Soloway, 1986).

4.4 People form and retrieve schemas when writing programs

Plenty of past research on how novices and experts program is based on cognitive psychology. Schema theory has been particularly influential in this work.

4.4.1 Plan schemas store general solution patterns

Expertise in programming, as in other fields, is marked by an improved knowledge base of generic solutions. Familiar problems for which a schema is available can be dealt with using more efficient strategies than unfamiliar problems, enabling experts to program effectively. Compared to expert programmers, novice programmers lack general schemas of 'canned solutions' to recurring problems and subproblems, which hinders problem solving.

Plans and plan schemas

Research on problem-solving schemas in programming was particularly active in the 1980s, when it revolved to a great extent around the pivotal and prolific figure of Elliot Soloway. Soloway and his colleagues used the term "plan" for the stereotypical action structures that programmers use as they design, write, and read programs. Consider the short program in Figure 4.2. Soloway (1986) presents an analysis of this program in terms of the plans that have contributed to its construction. Plans at various levels meet the subgoals of the programmer and combine to meet the overall goal of determining the average of given numbers. A running-total-loop plan has been used to compute the sum of input values. A counter-loop plan has been used to count the inputs. A division plan is used to compute a division. These three plans combine to meet the goal of computing the average. A print plan is used to print out the result. The running-total-loop plan and the counter-loop plans involve a stopping condition; a sentinel-controlled-loop plan caters for this subgoal. A skip-guard plan is associated with the division plan to prevent division by zero.

The example illustrates how, in order to produce a program, the programmer has applied schematic knowledge at many different levels of abstraction, from an overall program plan to lower-level plans to ever lower levels that are realized as just a single statement or expression. Plans are combined through abutment (one after the other, e.g., print after calculating), nesting (e.g., printing within the guard plan) and merging (e.g., the running-total-loop plan interleaves with the counter-loop plan) (Soloway, 1986).

The term "plan" requires a bit of clarification. Soloway and his colleagues used the word both as a term for generic problem-solving schemas within programming and to refer to specific instantiations of those generic patterns within solutions to particular problems. Rist (1989) makes note of this terminological issue – which recalls the 'broad vs. narrow schema' debate from Section 4.2 – and proceeds to use separate

terms for the two meanings. Following Rist's lead, I will use the term *plan schema* to mean an abstract problem-solving schema that mentally represents a generic solution to a generic problem, and *plan* to refer to specific solutions in specific contexts (which may be instances of a generic plan schema known to the programmer).

Plan schemas and programming expertise

There is considerable empirical evidence that plans and plan schemas are the basic cognitive chunks used in designing and understanding programs.

Many publications have demonstrated how researchers can analyze programs in terms of their constituent plans (e.g., Soloway et al., 1982, 1983; Spohrer et al., 1985; Soloway, 1986; Rist, 2004). A number of empirical studies support the idea that programmers make use of plan schemas when working on programming tasks (see, e.g., McKeithen et al., 1981; Soloway and Ehrlich, 1986; Soloway et al., 1988a; Rist, 1989; Détienne, 1990). These studies have not only shown that novices are limited by their relative lack of schemas but that experts, too, have difficulty with 'unplanlike' programs that do not follow the schematic patterns the expert is accustomed to. It has been argued that plan schemas are the single most important feature of the programming expert. On the basis of their empirical results, Spohrer and Soloway (1986a,b) contended that novices' lack of problem-solving schemas is a greater challenge for educators than misconceptions about particular language features.

A prominent thread of research investigating the relationships between plan schemas, strategies, and expertise runs through the CER literature. Immediately below, I will focus on the roles of schemas in program writing. Program comprehension will be dealt with in Section 4.6.

4.4.2 Programming strategy depends on problem familiarity (read: schemas)

From the literature, three related dichotomous pairs of program implementation strategies rise to the fore: top-down vs. bottom-up, forward vs. backward, and breadth-first vs. depth-first (Rist, 1989).

Top vs. bottom, forward vs. backward, breadth vs. depth

A *top-down* design strategy starts with an overall problem at a high level of abstraction and decomposes it into subproblems. The subproblems are further decomposed until at the lowest level of the hierarchy, solutions to subproblems are written as program code. Conversely, *bottom-up* design starts at a lower level of abstraction to produce pieces of the solution, which are then used as a basis for reasoning about and solving a higher-level problem, and joined together hierarchically until they fulfill the overall purpose of the program.

Forward development means producing a program in the order in which it appears in the program code. This is made possible by what Rist (1989) calls *schema expansion*: a previously known plan schema is activated that specifies the solution steps that are needed. It is instantiated as a specific plan, and applied to the particular situation so that the coding process reflects the order in which the solution is mentally represented in the plan schema. In *backward development*, on the other hand, the programmer goes back to earlier sections of code to make additions or modifications according to a plan that is created on the fly during coding.

A *breadth-first* strategy is one in which all the subgoals and solutions at one level of abstraction are dealt with uniformly before moving on to another level. For instance, a programmer using a top-down, breadth-first strategy would solve all the subproblems at one level of abstraction, decomposing them into smaller subsubproblems, before considering the solutions of the subsubproblems at an even lower level of abstraction. Contrast this with a programmer using a top-down, *depth-first* strategy, who first exhaustively solves a single subproblem by successively decomposing it into ever smaller problems until a concrete solution is found, and only then considers how to solve the other subproblems.

Early studies: mixed findings

The early literature on empirical studies of experts' and novices' programming strategies has been reviewed by Rist (1989; see also Visser and Hoc, 1990). I will summarize briefly.

Jeffries et al. (1981) concluded that both novice and expert programmers used a decompositional top-down strategy and forward development to create programs. The difference they found is that novices used a depth-first strategy that concretized a subsolution as program code as soon as possible, whereas experts used a breadth-first strategy that kept all the parts of the design equally abstract at any given time. Anderson et al. (1984) also concluded that novices used template-based strategies that can be characterized as top-down, forward-developing, and depth-first.

The difference between expert and novice programming strategies is not quite as straightforward as these early studies might suggest, however. Guindon et al. (1987) report a study of expert designers in which top-down design was rarely seen and expert strategies were instead characterized by exploratory and serendipitous behavior suggestive of a bottom-up strategy. Visser (1987) found that an expert may combine bottom-up and top-down strategies and shift between the two during the program authoring process. Rist (1989) pointed out that when writing a program, novices tend to flounder and search for a solution with little overall plan or organization. He observes on theoretical grounds that novice programmers cannot possibly always use top-down strategies since they simply do not have the abstract plan schemas that would allow a problem to be matched with a solution and decomposed into a set of connected pieces. This observation is supported by the prior finding that novices have only low-level representations of programming knowledge, whereas experts have representations at both the abstract and concrete levels.

The gist of Rist: mixed strategies

The work of Adelson and Soloway (1985) hints at a solution to these mixed findings. They found that when working with a familiar problem domain, expert programmers mentally simulated a solution at each successively lower level of abstraction according to a top-down, forward-developing, breadth-first strategy. However, in an unfamiliar domain where they lacked complete solutions, the experts reverted to a bottom-up strategy involving simpler local models which they tested before combining such pieces to form a full solutions.

Building on Adelson and Soloway's work, Rist set out to explain the partially conflicting results on programmer strategies. He presented a sophisticated, empirically supported theory of how people use and create plan schemas during programming. I will describe the theory in brief; for more detail, see the original publications (Rist, 1986, 1989, 1995, 2004).

According to Rist, people use top-down, forward-developing, breadth-first strategies to solve programming problems whenever they can, that is, whenever a problem is familiar and they have a suitable plan schema available. When confronted with unfamiliar or particularly difficult problems, people revert to bottom-up, backward-developing, depth-first strategies in order to develop new solutions. Since a problem and its solution may have parts that are familiar and others that are not, programmers alternate between the two kinds of strategies. When a plan schema can be retrieved for an overall problem on a high level of abstraction, it is used to produce a high-level solution in the order suggested by the schema. Then each of the subproblems is addressed at a lower level of abstraction, retrieving and using a plan schema for each, if possible. When a problem does not have a retrievable solution in memory, one needs to be created. This is where the programmer reverts to a bottom-up, depth-first strategy, forming solutions first at a very low, concrete level and combining them until the unfamiliar subproblem gets solved. This process is characterized by backward development. When the previously unfamiliar problem is solved, a new plan schema is formed and stored for later retrieval. With further practice, plan schemas become increasingly abstract and their use increasingly automatic.

An implication of Rist's theory is that none of the strategies are exclusive to novices or experts. The key difference is that experts have more plan schemas, which are furthermore more abstract and apply to a wider range of cases. It is precisely because experts have knowledge of more kinds of problems that they can rely more on top-down, forward-developing, breadth-first strategies. For a complete beginner, any programming problem will be unfamiliar and involve relatively slow, difficult and error-prone bottom-up, backward, depth-first development.

By means of a longitudinal study of novices' programming behavior, Rist (1989) found that as students gained experience with programs of a particular kind, they shifted from bottom-up backward development to top-down forward development. From an educational point of view, Rist's finding is very interesting.

It suggests that the growth of expertise is marked not by adding top-down strategies to one's arsenal, as has sometimes been assumed, but by being able to use top-down strategies as a result of growing familiarity with problem types and their solutions. Furthermore, Rist's theory explains other researchers' findings that suggest that expert programmers spend significant amounts of time on analyzing problem descriptions and planning what to do while novices tend to rush to concrete code and make local changes rather than work on the big picture (Adelson and Soloway, 1985; Linn and Dalbey, 1985; Robins et al., 2003, and references therein). It is worthwhile for experts to spend time on figuring out which schemas to apply, but novices' lack of schemas forces them to work bottom-up from a low abstraction level.

Variable-related schemas and roles of variables

Many plan schemas in programming are related to the use of variables. For instance, a simple programming schema serves to explain the use of variables as counters whose values start at zero and are then repeatedly incremented by one. Commonly, the ways in which a variable is used in a program are not defined by a single line of code or even by consecutive lines; references to each variable are spread throughout the program code. In the terminology of Letovsky and Soloway (1986), the plan for such a variable is *delocalized*. Delocalization of a plan increases the cognitive load of a programmer trying to comprehend it, since multiple separate units have to be kept in working memory simultaneously in order to figure out the plan (more on cognitive load in Section 4.5 below). Novice programmers may find this cognitive load very difficult to cope with.

Attempts to clarify delocalized plans have included documentation (Soloway et al., 1988b), software tools (Sajaniemi and Niemeläinen, 1989), and techniques that make plans explicit. An example of the latter line of work is that of Sajaniemi (2002) who studied the nature of variable-related plan schemas. He proposed a set of *roles of variables*, which capture stereotypical patterns of variable use; an example is the 'most-wanted holder', which stores a value that best matches a particular criterion amongst a number of candidates (e.g., the largest integer encountered so far in a loop). Sajaniemi concluded that 99% of the variables used in novice-level programs can be described using a small set of roles. He further showed that his role set can be identified in expert programmers' thinking and therefore can be said to be an explication of experts' tacit schematic knowledge (Sajaniemi and Navarro Prieto, 2005). Byckling and Sajaniemi (2006a,b) found that role-based teaching makes a difference to novice programmers' programming strategy: students taught using the roles of variables tended to use forward-development more than students taught in a more traditional way, suggesting an improvement in schema formation.

What about pedagogy?

The literature on programming strategies suggests that the formation of problem-solving schemas is a key challenge of programming education. Novice programmers need to form schemas at multiple levels of abstraction, from the basic building blocks to increasingly complex abstractions that 'put the pieces together'. It is through an improving knowledge base of schemas that the novice gradually becomes able to employ more sophisticated strategies and work on programs in an increasingly top-down, forward-developing way.

It has been suggested that novices' programming strategy can depend on the tools used; Meerbaum-Salant et al. (2011) report that Scratch – a visual programming environment for beginners (MIT Media Lab, n.d.) – fosters an extremely bottom-up approach to program writing in which novices put visual code components together with barely a thought for the big picture. On the other hand, Hu et al. (2012) report positive results from a pedagogical reform in which students were taught to explicitly analyze goals and form and merge plans in Scratch programs.

Many schema-theory-based recommendations for CS1 pedagogy call for plan schemas to be made explicit in instruction. The roles of variables, mentioned above, are one concrete initiative; for more, see Chapter 10 on CS1 teaching strategies. The importance of lower-level schemas of particular kinds of statements and expressions should not be underestimated either; I will return to this point in Section 5.6 below.

The implications of schema theory have been expounded on by cognitive load theory, discussed next.

4.5 Cognitive load strains the human working memory during learning

Cognitive load theory is a framework for investigating the relationships between the human cognitive architecture (from Section 4.1), schema formation (Section 4.2), and the structure of the information that one learns about.

The central theses of cognitive load theory are that people learn best when their working memory is not strained too much or too little, and when as much as possible of the working memory load is directed towards the formation of schemas. The requirements imposed on working memory – that is, one's cognitive load – depend on the structure of the information that needs to be processed, which may in turn be manipulated through instructional design.

What is now known as cognitive load theory originates in Sweller's (e.g., 1988) investigations of problem solving. It builds on earlier theories of working memory and schema acquisition, and on the concept of subjectively experienced 'mental load' from human factors science. The volumes edited by Paas et al. (2003) and Plass et al. (2010) give excellent introductions to cognitive load theory; my summary below draws especially on these sources.

4.5.1 Some cognitive load is unavoidable, some desirable, some undesirable

Traditional "triarchic" cognitive load theory divides working memory load into three components: intrinsic, germane, and extraneous.

Intrinsic cognitive load is imposed upon the learner's working memory by the material that is to be learned and other aspects of the learning task. Intrinsic load hinges on *element interactivity*, that is, the degree to which learning task involves interacting elements that must be held in working memory simultaneously in order to succeed. Paas et al. (2003) use an example from photo editing: learning to understand the interactions between changing color tones, darkness, and contrast requires the consideration of each element simultaneously. Different materials and learning goals differ in element interactivity. Element interactivity also crucially depends on the learner's existing schemas, which determine what constitutes an element. Intrinsic cognitive load is reduced by the learner's prior knowledge of the subject matter: as described in Section 4.1, a complex existing schema may be treated as a single chunk, reducing the demand on working memory. A schema automated through practice may not impose any cognitive load at all. Intrinsic cognitive load cannot be reduced without compromising the learning outcomes of the present learning activity.

Germane cognitive load refers to working memory usage that is non-essential in the sense that it is possible to carry out the task at hand without it, but that is nevertheless needed for learning. Germane load contributes to the creation and enhancement of schemas in long-term memory, and the automation of those schemas. For instance, the cognitive effort involved in noticing underlying similarities and principles in superficially dissimilar examples constitutes germane cognitive load. In other words, germane cognitive load is the cognitive load that is devoted to effortful learning.

Unnecessary content in learning materials, content that is not easy to access while solving a problem but needs to be kept in mind, disturbing background noise, redundant instructions that need to be scanned for new content, instruction that introduces unnecessarily many new topics at once, and verbose lists of examples of factors causing extraneous cognitive load which appear at the beginnings of sentences that fail to start by explaining what it is that they list, are examples of factors causing *extraneous cognitive load*. Extraneous cognitive load is non-essential load that is unhelpful for learning about what is intended to be learned about – and consequently often harmful. It is brought about by materials, instructional setups, and other environmental factors.

Within cognitive load research, the types of load have generally been considered to be distinct and additive, with the total cognitive load being the sum of the three components (Figure 4.3). Instruction should be designed so that germane cognitive load is high. This implies that the sum of intrinsic and extrinsic loads should not exceed working memory capacity. The theory warns that when intrinsic cognitive load is high (as a result of insufficient prior knowledge and – consequently – high element interactivity), the learner is likely to be overwhelmed by any extraneous load, and hard pressed to find capacity for germane load. On the other hand, when intrinsic load is low, extraneous load is less of an issue; however, the lack of germane cognitive load may still present a problem if the learner fails to put their mind to

work (because of lack of motivation, for instance). Both scenarios, which are illustrated in Figure 4.3, present challenges for learners and teachers.

While the threeway additive formulation of cognitive load theory may be overly simplistic – and has been questioned in the recent literature[2] – it serves to explain many of the interesting interactions between types of cognitive load that have been documented and is sufficient for my present purposes in this thesis.

4.5.2 Cognitive load theory has many recommendations for instructional design

Cognitive load theory has also contributed more specific points concerning the conditions under which meaningful learning is likely to take place. Researchers have found empirical evidence for a number of conclusions – often termed 'effects' or 'principles' – regarding the impact of cognitive load on learning. I summarize some of the main ones here; for more, see Plass et al. (2010) and references therein.[3]

Reducing extraneous load: the worked-out example effect and the completion effect

Some of the main findings of cognitive load theory concern learning by problem solving.

Problem-solving assignments are popular in education. However, cognitive load theory suggests that just solving problems is not the best way to learn to solve problems. Learning through problem solving is taxing for working memory, as resources have to be devoted to searching the problem for information relevant to solving it. Novices lack powerful schemas that suggest solution strategies, so they resort to weak generic strategies such as means-ends analysis, which require numerous elements to be kept in working memory at a time (see, e.g., Sweller, 1988; Paas et al., 2003). Little to no capacity is left for forming schemas in long-term memory to inform later problem solving. Some of this load is extraneous as it can be reduced by using different modes of instruction.

According to the *worked-out example effect*, extraneous cognitive load is reduced by studying *worked-out examples* of problems rather than solving the same problems oneself (see, e.g., Renkl, 2005). Worked-out examples are expert-produced solutions to a problem, often including explanations of the steps that were used to produce the solution. They are a staple of the cognitive-load-informed educational setting. Studying worked-out examples enables learners with limited schemas to allocate more of their cognitive capacity to germane load.[4]

A weakness of worked-out examples is that they may not engage the learner enough to induce germane load. The effectiveness of the examples is negated if the learner does not study them attentively and explain the given solution to themselves. The literature suggests that many learners are not naturally inclined towards effective spontaneous self-explanation of worked-out examples (see Renkl, 1997; Renkl et al., 1998; Renkl and Atkinson, 2003; Mayer and Alexander, 2011, and references therein).

Van Merriënboer et al. (e.g., 2003, and see Section 4.5.3 below) have noted a related *completion effect* according to which learning is enhanced by doing not full-blown problem solving but *completion problems* that provide the learner with a partial solution to be completed.

[2] Leading cognitive load theorists have recently come to emphasize the difficulty of telling apart intrinsic and germane cognitive load and to question the existence of germane load as a distinct source of cognitive load (Schnotz and Kürschner, 2007; Sweller, 2010b; Kalyuga, 2011). Along these lines, Sweller (2010b) and Kalyuga (2011) have advanced a reformulation of cognitive load theory which differentiates between two different aspects. The first aspect, *element interactivity*, corresponds to the working memory demands imposed by a learning task on a fully motivated learner (with a particular extent of prior knowledge) who utilizes their entire cognitive capacity. Overall element interactivity is the sum of intrinsic and extraneous element interactivity. The second aspect is the *actual working memory usage* of a (possibly unmotivated) learner when engaged in the learning task. This latter aspect depends on the former and also on factors such as motivation, which determine how or whether the learner actually processes the material at hand and what elements of the material they focus on. Working memory usage that is directed at processing intrinsic element interactivity is germane; other working memory usage is extraneous. I feel that this is a very useful clarification of the fundamental concepts of cognitive load theory for the future. However, in the present work, I remain with the better-established concepts of the traditional theory.

[3] I will discuss two more cognitive load effects, which are related to how information is presented – the modality effect and the split-attention effect – in the context of a visualization system in Chapter 15.

[4] The worked-out example effect, like most of the cognitive load effects, has been documented in the context of individual learning. Recent research on groupwork undertaken from a cognitive load perspective suggests that groups of novice learners can deal with more complex tasks than individuals, as their joint information-processing capacity exceeds that of the individual, despite the communication overheads (Kirschner, 2009). Some of the instructional recommendations from cognitive load theory may only hold for individual learning.

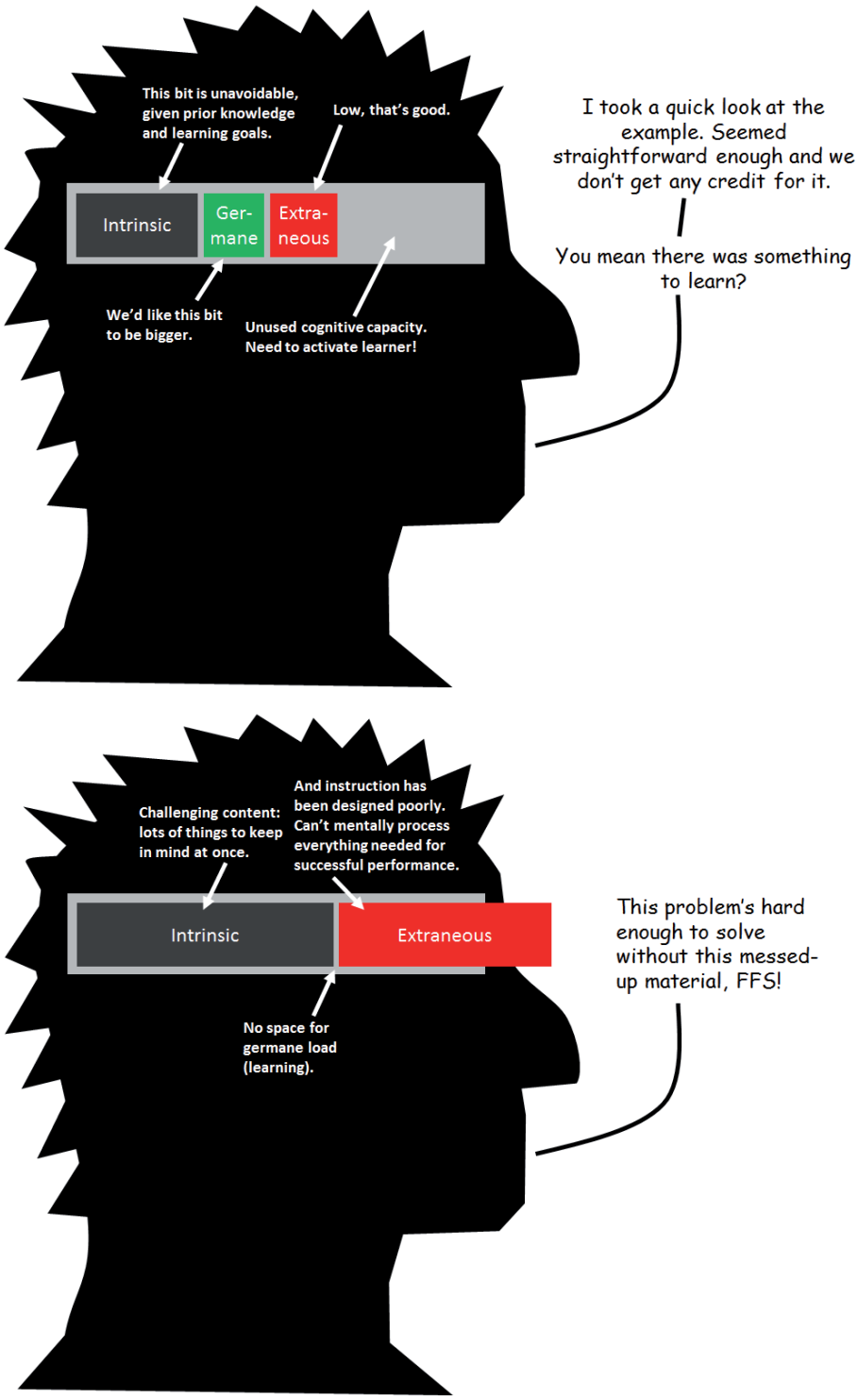


Figure 4.3: The different types of cognitive load in human working memory (as envisioned by traditional cognitive load theory). Two different scenarios are illustrated which have resulted in too little germane cognitive load.

The effect of prior knowledge: the guidance-fading, expertise reversal, and redundancy effects

Cognitive load theorists often recommend using multiple worked-out examples or completion problems as practice before a problem-solving task on the same topic. This emphasis on worked-out examples, completion problems, and the like does not mean that students should not solve problems. On the contrary, lower-load tasks serve to make problem-solving tasks better: they help in initial schema formation so that subsequent problem-solving tasks can be tackled effectively. "Guidance can be relaxed only with increased expertise as knowledge in long-term memory can take over from external guidance" (Kirschner et al., 2006, p. 80). According to the *guidance-fading effect*, decreasing support which takes into account learners' increasing experience is superior to worked-out examples alone (see, e.g., Renkl, 2005). With reference to the four-stage model of skill development of Anderson et al. (1997), Atkinson et al. (2000) have argued that worked-out examples are likely to work best in the early phases of the development of a cognitive skill, during which learners rely on analogies, have little practice, and are only starting to construct abstract problem-solving rules.

The guidance-fading effect is a manifestation of the more general phenomenon of the *expertise reversal effect*, which states that learning activities that are suitable for novices may become ineffective, or even harmful, as experience grows. Conversely, activities that are unsuitable for novices are often suitable for more advanced students. Several of the other cognitive load effects have been shown to be dependent on expertise (Kalyuga et al., 2003; Kalyuga, 2005, and references therein).[5]

Part of the reason why the expertise reversal effect occurs is that guidance given to experts may in fact hinder them, as they have to relate and compare the (to them unnecessary) guiding framework to their existing knowledge to find out what is relevant. This is an example of the yet more general *redundancy effect*, which states that presenting unnecessary information interferes with learning (see, e.g., Sweller, 2005).

Portioning intrinsic load: the isolated/interacting elements effect

The *isolated/interacting elements effect* states that "learning is enhanced if very high element interactivity material is first presented as isolated elements followed by interacting elements versions rather than as interacting elements form initially" (Plass et al., 2010, p. 30). This is essentially an endorsement of a part-whole approach to learning in which learners initially deal with small parts of a topic, and are not subjected to the complexity of the big picture until they have formed rudimentary schemas, however incomplete, of the parts.

Part-whole approaches run the risk of resulting in poorly integrated piecemeal knowledge, but may be worth the risk in the case of very highly interactive material, assuming that care is taken to foster the integration of the parts initially learned.

Increasing germane load: the variability effect and the imagination effect

Two cognitive load effects deal in particular with how certain kinds of activities can increase germane cognitive load.

The *variable examples effect* states that examples whose surface features are dissimilar to each other are more effective than examples that look similar on the surface (Paas and van Merriënboer, 1994). The variation encourages learners to identify the similarities between situations and thereby helps with the creation of generic schemas that are suitable for transfer.

The *imagination effect* states that having learners imagine themselves carrying out procedures related to worked-out examples enhances learning compared to having them simply carefully study the examples for the same length of time. The imagination task encourages learners to make use of their available cognitive capacity to mentally practice relevant skills (Cooper et al., 2001). However, to be effective, imagination appears to require significant working memory resources available for germane load (that is, the mental manipulation of and learning from the intrinsic content). Consequently, the imagination effect

[5] The expertise reversal effect is not a novelty discovered by cognitive load theorists. As discussed by Plass et al. (2010), the studies on expertise reversal are effectively an extension of the aptitude–treatment interaction studies carried out since the 1970s to examine the relationships between instructional methods and students' personal characteristics.

depends on prior knowledge in accordance with the expertise reversal effect: the literature suggests that imagination tasks work best for learners with some experience and existing schemas, and poorly if at all for beginners (Cooper et al., 2001; Kalyuga et al., 2003). On a related note, Renkl (1997) found that anticipating solution steps while studying a worked-out example was an effective way of learning, but only when the learner already had some prior knowledge.

Taking the effects into account

Van Merriënboer et al. (2003; van Merriënboer and Kirschner, 2007) present a general model for the design of learning environments that is mindful of the various cognitive load effects. In their model (dubbed 4C/ID) students are given learning activities that are divided into "task classes" of increasing complexity. Within each task class, the nature of the material covered is the same (that is, each activity has the same element interactivity and relies on the same set of generic knowledge in the form of schemas). To combat extraneous cognitive load while also catering for the expertise reversal effect, guidance fades within each task class: learning within each task class begins with a strongly guided activity such as studying a worked-out example, and ends with a considerably more challenging task such as solving a problem. Intermediate activities may include completion tasks, for instance. As a result of the simple-to-complex sequencing of task classes, the formation of schemas within a task class serves to reduce intrinsic load as the learner tackles the next task class on the same topic. To increase germane cognitive load, the tasks within each task class are designed to be variable in their surface features. To avoid overload during task performance, information about the general principles that pertain to the task class is presented to students before they tackle the activities of the task class (and is kept accessible throughout). However, procedural information that helps with specific learning activities is integrated into the learning environment and presented in a just-in-time fashion during the activities. Finally, part-task practice ('etudes' or 'drills' on a decontextualized aspect) may be used for selected parts of complex tasks for which the learners urgently need automated schemas.

The 4C/ID model is in many ways representative of cognitive load theorists' advice on instructional design. It underlines the need for deliberate practice and carefully designed guidance. It calls attention to the efficacy of activities other than problem solving on the way towards building problem-solving ability. It attends, via a managed learning environment, to the interplay between the element interactivity inherent in the content of learning and the learner's prior knowledge.

The design of learning environments is traditionally in the hands of the teacher. The impact of learners' personal goals on cognitive load is an open question that has not been much discussed in connection with cognitive load theory (Gerjets and Scheiter, 2003). Cognitive load theory does, however, highlight challenges in the use of highly learner-controlled learning environments. For instance, can a learner make reliable judgments on the element interactivity of content they are yet to learn about?

4.5.3 Some effects of cognitive load on introductory programming have been documented

There is a small body of cognitive-load-related research on the learning of introductory programming. Shaffer et al. (2003) discuss the potential of cognitive load theory in computing education, review earlier work, and observe that "the connection between cognitive load theory and the challenges faced by novice computer science students has not been fully addressed".

Van Merriënboer (1990; van Merriënboer and de Croock, 1992) found empirical support for the completion effect in the context of introductory programming education. In two separate experiments, students who modified and extended existing programs learned to write programs better than did others in a control group that wrote programs from scratch. In a similar study with third-year students as subjects, Nicholson and Fraser (1996) found no similar effect; Shaffer et al. (2003) speculate that the level of prior knowledge of programming may be the reason behind these mixed findings.

A study of the impact of reading assignments was conducted by Linn and Clancy (1992), who compared the performance of novices who designed and wrote their own programs to that of other novices who instead studied expert commentaries on how to solve the same problem (worked-out examples, in other words). Linn and Clancy conclude that such expert commentaries are eminently useful, and that in fact

"writing a computer program is less helpful than having expert commentary for developing design skills, even when test questions require application of templates that were used for writing the program".

The results from these studies are in line with cognitive load theorists' recommendations to use worked-out examples and completion problems in addition to problem-solving tasks. These activities emphasize the pedagogical value of reading code, as opposed to merely designing and writing it. In this respect, cognitive load theory is compatible with the analyses of programming courses reviewed in Chapter 2, according to which the goals of introductory programming courses are cognitively demanding, and with the BRACElet project's hypotheses as to how reading and writing skills are interdependent in terms of their development (Chapter 3).

Robins (2010) discusses a related issue as he presents his hypothesis of why learners fail to cope with introductory programming courses. One of the underlying reasons, Robins argues, is that basic programming concepts are particularly densely connected to each other. In other words, introductory programming has an unusually high intrinsic cognitive load. A related point was made earlier by du Boulay (1986), who observed that programming is difficult to learn because the novice needs to deal with multiple interwoven challenges at once: programming notation, runtime dynamics, the need to mentally represent programs and their domain, tool use and the programming process, problem-solving schemas, and the notion of programming in general.

The theory of threshold concepts (more on which in Chapter 9) suggests that certain perspectives are particularly challenging to develop, in part because of the way they require multiple concepts to be integrated. Such learning thresholds may involve particularly high intrinsic cognitive load. Program dynamics, information hiding, and object interaction have been proposed as thresholds in introductory computer programming.

As noted, advice on pedagogy that draws on cognitive load theory tends to emphasize the need for teachers to carefully design direct, explicit guidance for learners (as opposed to student-directed, free exploration of complexity). The advice concerning introductory programming pedagogy is no different. I will return to cognitive-load-inspired pedagogies for CS1 in Chapter 10.

Cognitive load theory highlights how useful it is for novices to read plenty of program code. This brings us to the topic of the last section in this chapter, program comprehension studies.

4.6 Both program and domain knowledge are essential for program comprehension

A *program comprehension model* is a theoretical model of the mental representations that a programmer forms as they familiarize themselves with a program. A program comprehension model may describe the formation, use, structure, and/or content of those mental representations. Many different program comprehension models have been proposed in the psychology of programming literature since the 1970s, usually drawing on empirical investigations of novice and/or expert programmers. My review in this section draws on the original studies cited below and on previous reviews of program comprehension models, in particular those by von Mayrhauser and Vans (1995), Corritore and Wiedenbeck (2001), and Robins et al. (2003); see also Schulte et al. (2010). I comment primarily on three prominent themes within the program comprehension literature: the distinction between program and domain models, the distinction between top-down vs. bottom-up comprehension strategies, and the effects of programming paradigm on program comprehension.

4.6.1 Experts form multiple, layered models of programs

Program models and domain models

At least since the early program comprehension model presented by Brooks (1983), most program comprehension models have posited two components within people's mental representations of programs:

a program model and a domain model (in addition to various other structures posited by different theorists). These constructs are sometimes called by different names, and the definitions differ in their details, but the basic concepts are widely shared. A *program model* is a mental representation of the program itself such as the program text, the elementary operations the code performs (e.g., assignments, function calls), and the control flow of the program. A *domain model* is a mental representation of the problem that the programmer is trying to solve and its solution. It contains knowledge about the 'world of the problem', e.g., its components, their relationships, the data flow between components, and the purpose of the program overall, as well as the goals of its subprograms. Both program and domain models can contain knowledge of both static and dynamic aspects of a program.

The division between program and domain models in CER was influenced by the text comprehension studies of van Dijk and Kintsch (1983), who distinguished between a *textbase* (a representation of the semantics of a text that has been read) and a *situation model* (a representation of the world that the text is about). Program and domain models are not schemas in the sense of general knowledge, but their formation is guided by schemas and they incorporate elements of instantiated schemas.

Top-down or bottom-up? (The answer is in schemas.)

In Section 4.4, we saw a mix of early research results suggesting top-down program authoring strategies on the one hand and bottom-up strategies on the other. A similarly mixed set of findings characterized early program comprehension studies. Consequently, some early models emphasized the role of hypotheses of program behavior at a high level, recognizing common patterns in code and applying the corresponding plan schemas in a *top-down* way to make sense of a program (Brooks, 1983; Soloway and Ehrlich, 1986). Others found evidence for a *bottom-up* view of program comprehension, where a detailed program model is formed before a more abstract domain model (Pennington, 1987a,b; Corritore and Wiedenbeck, 1991; Bergantz and Hassell, 1991).

In program authoring studies, Rist's theory of schema formation (Section 4.4.2 above) explains how programmers mix top-down and bottom-up strategies depending on the availability of retrievable plan schemas. Similarly, researchers have presented program comprehension models according to which comprehenders use a mix of top-down and bottom-up strategies in opportunistic ways (von Mayrhauser and Vans, 1995; Corritore and Wiedenbeck, 2001, and references therein). When programmers seek to understand a program in an unfamiliar domain, they tend towards bottom-up comprehension strategies in which they first construct a program model. The program model allows the programmer to reason about the program's control flow, and is used to develop a domain model that deals with data flow and the purpose of program components. A top-down approach is used when constructing hypotheses about program behavior is possible, that is, when the domain is familiar enough. The direction of comprehension may change frequently during a comprehension process if parts of the program differ from each other in terms of familiarity or difficulty.

Wiedenbeck et al. (1993) studied the characteristics of experts' and novices' mental representations of programs and concluded that experts' mental representations are hierarchical and multilayered, contain explicit mappings between the different layers, are founded on the recognition of basic patterns (schemas), and are well connected internally and well grounded in the program text. Novices' mental representations exhibited these characteristics only to a much lesser extent. As in the case of program writing, experts can rely more on top-down comprehension strategies since they are familiar with more domains, problems, and solutions.

Paradigm matters for comprehension strategy

Research suggests that the programming paradigm used can make a difference in program comprehension. In particular, the paradigm influences the order in which program and domain knowledge is formed during the comprehension process.

Within the paradigms of procedural programming (Pennington, 1987a,b; Corritore and Wiedenbeck, 1991) and logic programming (Bergantz and Hassell, 1991), the literature suggests that when studying a program, programmers tend to construct first a program model, then a domain model. Results from object-oriented programming have been different, however. Burkhardt et al. (1997, 2002) extended Pennington's

model of program comprehension to object-oriented programming. They found that people studying object-oriented programs formed a domain model early on during the comprehension process. This, the authors argue, is due to the nature of object-orientation, which emphasizes modeling the domain as objects and classes at the expense of program model aspects such as control flow. The results of Corritore and Wiedenbeck (2001) point in the same direction, and also suggest that object-oriented experts tend to use top-down strategies more than procedural experts do. Khazaei and Jackson (2002) found that, from the program comprehension perspective, event-driven programming resembles OOP rather than procedural programming in its emphasis on domain knowledge.

4.6.2 Novices need to learn to form program and domain models

Let us now consider program comprehension from the perspective of introductory programming education. Some research results point at a learning hierarchy of program comprehension skills, which, interestingly, may be different in different programming paradigms.

Paradigm matters for learning to comprehend

Corritore and Wiedenbeck (1991) studied novices' comprehension of procedural programs. Their study suggests that novices first learn to analyze programs in terms of program models, and later learn to develop domain models as they gain expertise. That is, the order in which people learn to form each model is the same as the order in which they form these models while studying a particular program, which was discussed above. Follow-up work on procedural programming tends to support Corritore and Wiedenbeck's findings: Wiedenbeck and her colleagues found that when studying procedural programs, novices formed mental representations that were stronger in terms of detailed program knowledge than in terms of knowledge of program function (Wiedenbeck and Ramalingam, 1999; Wiedenbeck et al., 1999).

Some work has been done to investigate the relationship between the program and domain models of novice programmers taught in an object-oriented way. Wiedenbeck and Ramalingam (1999) studied students taking their second course in programming. They found that when reading short object-oriented programs, the students developed mental representations that were strong in terms of knowledge of program function (the domain model) but weaker in terms of detailed program knowledge (the program model). This is in contrast to the models that the same students constructed when studying procedural programs, where the relative strengths and weaknesses of the mental representations were the opposite. More specifically, Wiedenbeck and Ramalingam report that the novices who performed better overall did equally well with both programs of both paradigms, with a similar pattern of errors in both program and domain knowledge. However, the novices who performed worse were different: the lower-performing half developed a better program model than domain model of the procedural programs they read, but a better domain model than program model of the object-oriented programs. In another paper, Wiedenbeck et al. (1999) compare novices taught using the object-oriented paradigm to novices taught procedurally. Both groups performed more or less equally well when reading short programs, but with different patterns of errors.

One plausible interpretation of the results of Wiedenbeck and her colleagues is that object-orientation changes the learning path of novice programmers: learning proceeds from being able to construct a domain model towards being able to construct a program model. This contrasts with the results from procedural program comprehension studies.

Pedagogical implications

Recent work by Schulte (2008; Schulte et al., 2010) has sought to translate the psychology of program comprehension into educational practice. This initiative is timely, as despite there being a sizable body of theoretical work on program comprehension, few educators have drawn substantially on this work (as reviewed by Schulte et al., 2010).

Schulte et al. (2010) suggest that CS1 teachers should make it their business to foster students' ability to glean multiple kinds of information from reading a program, to form connections between pieces of program-related information, and thereby to develop a holistic understanding of the program. They

further point out that the direction of the teaching and learning sequence is a pedagogical choice: should one start with learning about programs' structure in terms of code and its runtime dynamics (to build a program model) or does one initially rely on domain knowledge? This question can be considered from the point of view of what is known about learning to comprehend programs in different programming paradigms. Where novices initially (naturally?) focus on building a program model – as seems to be the case in procedural programming – does this mean that they have less trouble with that aspect and teaching should focus on helping them form a domain model? Conversely, do object-oriented novices need additional help with the program model, as their paradigm primarily stresses the domain model?

Existing research does not provide unambiguous answers to these questions. Perhaps the most important message that we can take from program comprehension studies at the present time is that any introductory programming course should – in one way or another – teach students to build both program models and domain models when reading programs, and to relate the two.

Chapter 5

Psychologists Also Say: We Form Causal Mental Models of the Systems Around Us

> *In interacting with the environment, with others, and with the artefacts of technology, people form internal, mental models of themselves and of the things with which they are interacting. These models provide predictive and explanatory power for understanding the interaction.* (Norman, 1983)

In this chapter, I stay with cognitive psychology as I turn to theories of mental models and their implications for learning to program.

Section 5.1 below outlines the general properties of what are known as mental models. Section 5.2 elaborates on what the literature has to say about mental model formation and quality. Section 5.3 introduces the notion of a 'conceptual model' that may be employed as a pedagogical device for learning about a system. Section 5.4 brings us to programming, and one of the pivotal concepts of this thesis: the 'notional machine', that is, the runtime mechanism which novice programmers must learn to control via programming, and which they need effective mental models of. Section 5.5 discusses the challenges of the step-by-step tracing of programs, a key skill that novices struggle with, in part because of poor mental models of the notional machine.

In Section 5.6, I ask whether we should be more concerned about the lack of problem-solving skills and schemas, about misconceptions concerning fundamental concepts, or about the lack of tracing ability. To foreshadow my conclusion, each issue is important, and there is an argument to be made that understanding the role in program execution of the computer – the notional machine – is fundamental to each of them.

The final section, 5.7, is more generic. I wrap up the review of theories of cognition from the previous chapter and this one by pointing out and briefly discussing some criticisms of cognitivist approaches to educational research.

5.1 Mental models help us deal with our complex environment

Schema theory deals with the growth of expertise and the forming of generic knowledge through experience. Various scholars (e.g., Preece et al., 1994; Brewer, 2002) have argued that the generic concepts embodied in schemas are not sufficient in themselves to explain how humans deal with objects and systems – including ones that are unfamiliar to them. This is where mental model theory steps in.

A *mental model* is a mental structure that represents some aspect of one's environment. A mental model is often about a specific thing or system rather than a generic concept. For instance, I have mental models of myself, of my wife, and of the TeXlipse software system that I am using to write this text. These mental models, which I have formed in part consciously and in part unconsciously, allow me to reason about how these specific aspects of my environment function in different circumstances. The concept of mental model has been applied to various disciplines, such as human–computer interaction, physics, the design of everyday objects, computer programming, ecology, and astronomy (see, e.g., references in Rouse and Morris, 1986; Schumacher and Czerwinski, 1992; Markman and Gentner, 2001). In particular, mental model theorists have been interested in the interactions between humans and causal systems.

5.1.1 We use mental models to interact with causal systems

Causal vs. logical mental models

It is necessary at this point to differentiate between two influential threads of research on constructs called mental models.

The term "mental model" received wide recognition after its introduction to cognitive psychology in the early 1980s. The two main threads of research on mental models are often attributed to the near-simultaneous publication of two books titled *Mental Models*, which elaborated in two different ways on Craik's (1943) earlier notion of "models of reality". The articles in the volume edited by Gentner and Stevens (1983) greatly influenced research into the kinds of mental representations people store about physical and software systems. The book by Johnson-Laird (1983) is seminal to a body of work that investigates a certain kind of situation-specific mental representation that people create in working memory to help them reason about logical problems. Markman and Gentner (2001) term mental models as described in the Gentner and Stevens book *causal mental models* and those in Johnson-Laird's research tradition *logical mental models*. Logical mental models are not of interest for my present purposes; whenever I write about "mental models" in this thesis, I refer to causal mental models.[1]

Mental models of systems

People form causal mental models of all manner of things. Schumacher and Czerwinski (1992) note that while one can have a mental model of a marriage or a social environment, not all topics are equally well represented on the research agenda. Much of the research on mental models has focused on the way people interact with and think about complex physical and software systems that involve causal mechanisms (e.g., an electrical circuit, a word processor). The mental model is a theoretical construct posited to explain how people describe the purpose and underlying mechanisms of such systems to themselves and how they predict future system states. This emphasis stems partially from arbitrary historical reasons that arise from research tradition[2] and partially from practical reasons: "We would argue that experts in computer systems are easier to define than experts in marriage." (Schumacher and Czerwinski, 1992)

Research does exist on causal mental models that is not concerned with humans' relationships with technical systems. However, I will limit my discussion to causal mental models of technical systems.

Researchers have further sought to identify the features that mental models of systems have in general, the characteristics that distinguish between useful and not-so-useful mental models, the transferability of the knowledge stored in mental models, and the relationships between learning and mental model construction. I will comment on each of these topics below.

5.1.2 Research has explored the characteristics of mental models

According to Norman's (1983) seminal description, mental models:

- reflect people's beliefs about the systems they use and about their own limitations, and include statements about the degree of uncertainty people feel about different aspects of their knowledge;
- provide *parsimonious*, simplified explanations of complex phenomena;
- often contain only *incomplete*, partial descriptions of operations, and may contain huge areas of uncertainty;
- are *'unscientific'* and imprecise, and often based on guesswork and naïve assumptions and beliefs, as well as "superstitious" rules that "seem to work" even if they make no sense;

[1] The body – or rather, bodies – of research on mental models make up a complex terminological and conceptual tangle. As with the word "schema" (Section 4.2), some authors use "mental model" as an umbrella term for all knowledge, a practice that has been criticized by authors such as Rouse and Morris (1986). There are numerous other more or less idiosyncratic ways of using the term in the literature, which I will not cover here.

[2] The focus on causal systems and devices can be traced back to human–machine interaction studies in the 1960s and a body of research that investigates process control: the manual control of complex machines and, later, the supervisory control of increasingly automatic machines (Rouse and Morris, 1986; Wickens, 1996, and references therein).

- are commonly *deficient* in a number of ways, perhaps including contradictory, erroneous, and unnecessary concepts;
- *lack firm boundaries*, so that it may be unclear to the person exactly what aspects or parts of a system their model covers – even in cases where the model is complete and correct;
- *evolve* over time as people interact with systems and modify their models to get workable results;
- are *liable to change* at any time; and
- *can be 'run'* to mentally simulate and predict system behavior, although people's ability to run models is limited.

People commonly confuse or combine mental models of similar systems with each other. One may also have multiple mental models of a single system. Multiple models may cover different parts of the system in a non-overlapping and complementary way, or they may be parallel – perhaps contradictory – models of the same parts. Parallel models may also operate on different levels of abstraction with each other (e.g., one model describes the physical aspects of a system and another its functional purpose).

Mental models evolve in long-term memory. When we reason about an object or a situation, we use a mental model of it; when necessary, we construct new mental models. Schumacher and Czerwinski distinguish between studying *stable* mental models that predate the need to use them and *derived* mental models, which are created when a situation calls for them. A derived mental model can be stored or forgotten right after it has been processed. If we have related schemas, activating them contributes to the kind of mental model we construct as our prior generic knowledge affects the way in which we see new instances. Conversely, aspects of the mental models of similar instances may be abstracted into a more general schema. Metaphors and analogies can also play a part in constructing and evolving a mental model (see, e.g., Gentner and Gentner, 1983; Schumacher and Czerwinski, 1992). Mental models are often not the product of deliberate reasoning; they can be formed intuitively and quite unconsciously.

Like schemas, mental models have been argued to be part of what sets the expert apart from the novice. Experts rely on analogies based on existing mental models as they encounter new situations that require them to form new models – as novices do. However, experts' mental models are robust, based on a principled understanding of system components, and allow for unanticipated situations to be dealt with (de Kleer and Brown, 1981). Because uncertainty about system capabilities can lead to trying out multiple approaches, novices tend to rely more on multiple inconsistent causal mental models of systems, while experts are less likely to do so (see Schumacher and Czerwinski, 1992, and references therein). Compared to the *ad hoc* naïve models often employed by novices, experts' mental representations are relatively stable as the result of lengthy experience.

What does a mental model consist of? Researchers vary in how they describe the internal structure of mental models, with some emphasizing their role as collections of knowledge, others the role of metaphors and analogies, and yet others the role of procedural knowledge. Ways of describing the structure of mental models include "topologies of device models" (de Kleer and Brown, 1983) and "homomorphs of physical systems" (Moray, 1990). On the other hand, many authors do not concern themselves with the internal structure of mental models. Jonassen and Henning (1996) argue that mental models are inherently epistemic – that is, the mental models themselves affect how we understand and express them – and that therefore their actual contents are not readily knowable by others. Still, Jonassen and Henning continue, researchers can gain information about these models by eliciting people's structural and procedural knowledge of systems, as well as the metaphors and visualizations people use. For the purposes of the present thesis, the internal structure of mental models is not important.

5.1.3 Mental models are useful but fallible

> *One of the reasons that mental models are so important is that for the individuals who hold them, those models have a value and reality all their own. Individuals believe in them, often without direct reference to their accuracy or to their level of completeness, and are reluctant to give them up.* (Westbrook, 2006)

Mental models allow users to be comfortable with complex systems. Norman (1983) suggested that people rely on their mental models to develop behavior patterns that make them feel more secure about how they interact with systems, even when they know what they are doing is not necessary. Markman (1999, p. 266) further points out that people do not assess their mental models for completeness, but are content with partial knowledge and may not notice the limits of what they know. Mental models need to be only minimally viable to be maintained, and they do not even need to be accurate for some system users to feel they are fully satisfactory – an "ignorance is bliss" approach, as Westbrook (2006) puts it.

While mental models are clearly useful, they are also potentially dangerous. An inaccurate mental model will lead to mistakes. Kempton's (1986) research contributes the example of the thermostat: mental models based on ill-fitting analogies to, say, car accelerators, lead people to think that turning the thermostat up to 'full throttle' will heat the home faster. A poor mental model of a computer programming environment will result in bugs. Even though people themselves do not require their mental models to be complete and accurate in order to be used, they "certainly function with varying levels of efficiency and effectiveness as they employ mental models that are inaccurate and/or incomplete" (Westbrook, 2006). A further complication is that although models can be developed or corrected through practice and instruction, people often cling to emotionally comfortable and familiar existing models.

'Running' a model

According to Norman's description above, mental models are 'runnable'. This means that people can use mental models to reason about systems in particular situations, to envision with the mind's eye how a system works, and to predict the future (or past) behavior and states of a system given a set of initial conditions (see, e.g., de Kleer and Brown, 1981, 1983; Markman and Gentner, 2001). For instance, people can run their mental models to predict the trajectories of colliding balls in a physical system, the behavior of an existing software system under given parameters, or the behavior of a computer program which they are presently designing.

Running a mental model of a system is often called *mental simulation* of the system. I will use this term below.

Mental simulation is performed in working memory. It often involves visual imagery and may have a motor component. Since working memory capacity is very limited, it comes as no surprise that researchers have found that mental simulations involve only a very small number of factors. According to Klein (1999), for instance, even experts' mental simulations rarely involve more than three factors (or "moving parts") and six transition states (stages). Simulating a system's behavior at a low level of abstraction can fail as a result of too many variables or states.

Researchers (e.g., de Kleer and Brown, 1981; Markman and Gentner, 2001) have emphasized the often qualitative nature of mental simulations, meaning that simulations tend to be based on relative properties rather than specific quantities. People do not calculate the specific values of the variables involved in a simulation. Rather, they reason about relative properties such as relative speed, and relative mass in a physical system. Qualitative simulation does not require the significant computation that would be necessary to carry out detailed quantitative simulations. Since the simulation process is demanding, people shift to using learned rules and cached results as they gain experience with the system.

While the level of abstraction must not be too low, simulation at an excessively high level of abstraction will not produce working solutions to problems either, and even when the overall level of abstraction is appropriate, people tend to neglect or abstract out important information. To solve problems successfully, it is crucial to simulate systems at a level of abstraction that is just right for the problem at hand, and to focus exactly on those factors that are important to produce the kind of prediction or solution aimed for. To do so is difficult and requires considerable experience (Klein, 1999; Markman and Gentner, 2001).

5.2 Eventually, a mental model can store robust, transferable knowledge

In this section, I describe three different theoretical frameworks that give insights into the transferability of the knowledge stored in mental models. Schumacher's theory describes mental model formation as a three-stage sequence that culminates in abstraction from specific models to generic knowledge at the

expert stage. De Kleer and Brown characterize mental models in terms of their adherence to certain robustness principles, and suggest that robust models allow better transfer to new situations. Finally, Wickens and Kessel's studies show that certain kinds of learning activities are better than others at fostering the creation of mental models that transfer to similar tasks.

Schumacher: stages of model formation

Schumacher proposed a theory of forming mental representations of causal systems, which is empirically based and consistent with a number of related theories (Schumacher, 1987; Schumacher and Czerwinski, 1992). A three-stage sequence describes the acquisition of both specific and generic knowledge, with the latter arising from the recognition of common features in the former.

1. During the *pretheoretic stage*, an initial mental model of a specific system is formed by a user of the system. The initial model is a collection of retrieved experiences of superficially similar systems (e.g., other systems having a similar physical appearance). If no such experiences are found in memory, performance in using the system is reduced.

2. During the *experiential stage*, some understanding of causal relationships emerges through prolonged exposure to the system, even if the understanding is not supported by any superficial similarities to other systems. Knowledge embodied in the mental model is not yet readily transferable to other systems unless they are superficially very similar to the known system. During the experiential stage, the mental model becomes increasingly well ingrained.

3. During the *expert stage*, generic information is abstracted from multiple system representations. The user easily recognizes systemic patterns of behavior and effortlessly retrieves old knowledge about systems. Knowledge is easily transferred across instantiations of a system type, even when the system instances are superficially dissimilar.

This learning sequence provides a platform for making several points.

First, since previously known instances are used as a basis for learning, prior knowledge plays a key role in the acquisition of knowledge, both about specific systems and when abstracting to the general case at the expert stage.

Second, the early stages of learning about a system depend greatly on the superficial characteristics of systems. Many writers on mental models have noted the role played in mental model formation by social norms and schemas related to the surface structure of the system. Similar GUI controls in software applications create superficial similarities between systems, for instance.

Third, the experiential stage, which is often quite long, highlights the difficulty of transferring knowledge from one mental model to another. According to Schumacher and Gentner (1988), the less superficially similar two systems are, the worse the transfer is, even when the systems are functionally isomorphic. It is unrealistic to expect the transfer of a mental model before it is well ingrained.

Fourth, it is commonly easier to form some kind of pretheoretical mental model based on surface similarities than it is to substantially alter one's existing model during the experiential stage. As mentioned earlier, people tend to cling to their existing models. Moray (whose work is reviewed by Schumacher and Czerwinski, 1992) found that changing one's mental model took significantly longer than it took to originally form an initial model of similar complexity. Making matters worse is that mere coincidences can reinforce people's confidence in their existing yet flawed models (Besnard et al., 2004).

De Kleer and Brown: robustness

De Kleer and Brown's (1981; 1983) work on mental models provides another perspective on knowledge transfer. Using electrical devices as examples, they studied the mental models that people construct. De Kleer and Brown stressed the importance of knowledge that can be used to understand various systems, as "any single device however complex is of no fundamental importance". They argued that to be as useful as possible, a mental model should be internally consistent and correspond behaviorally to the actual device under consideration. Further, they argued that only certain kinds of mental models,

meeting certain "esthetic principles", lend themselves to answering unanticipated questions or predicting the consequences of novel situations. They contended that using such mental models, which they termed *robust*, is characteristic of expert behavior.

De Kleer and Brown's theory defines mental models as topologies of submodels that represent components of the system. Each submodel is a collection of rules that describe the causal behavior of a component. Robustness arises out of the component models. The better a mental model's component models meet the following principles, the more robust the overall model is.

- The *no-function-in-structure principle*: the rules that specify the behavior of a system component are context free. That is, they are completely independent of how the overall system functions. For instance, the rules that describe how a switch in an electric circuit works must not refer, not even implicitly, to the function of the whole circuit. This is the most central of the principles that a robust model must follow.
- The *locality principle*: the rules that specify the behavior of a system component are represented only in terms of the internal aspects of the component and its connections to other components, not in terms of the internal aspects of other components. For instance, the rules that describe a switch in an electrical circuit must not depend on the internal state of any other component in the circuit. The locality principle helps ensure that the no-function-in-structure principle is met.
- The *weak causality principle*: the rules of the mental model attribute each event in the system to a direct cause. The reasoning process involved in determining the next state does not depend on any "indirect arguments". For instance, what happens next to a component in an electrical circuit must be directly attributable to some local cause rather than indirectly inferred by elaborately reasoning about other components. The weak causality principle is important for the efficient running of the mental model.
- The *deletion principle*: the mental model should not predict that the system will work properly even when a vital component is removed.

An overarching aspect of robust models is that the components of the model are understood in terms of general knowledge that pertains to those components rather than specific knowledge that pertains to the particular configuration of the components. A non-robust model may serve for mental simulations of a particular system under normal circumstances. A robust model is needed for transferring the knowledge embodied in a mental model to a similar but novel problem. A robust model is also needed to mentally simulate exceptional situations such as when a component malfunctions or a change to the system is either made or planned. This makes robust models highly desirable.

Wickens and Kessel: transferable models through active learning

Wickens and Kessel's work (see Kessel and Wickens, 1982; Wickens, 1996; Schumacher and Czerwinski, 1992) provides another perspective on mental model formation. They studied the performance of people trained alternatively as *monitors*, who supervise a complex technological system, or as *controllers*, who control the system manually. As one would expect, Wickens and Kessel found that training in system monitoring improves people's monitoring skills, and training in controlling a system improves controlling skills. However, and significantly, they also found that the controllers could transfer their skills to monitoring tasks, while the reverse was not true of the monitors. The controllers were also found to be better at detecting system faults from subtle cues that escaped the attention of the monitors. Wickens and Kessel inferred that the two kinds of training led to different kinds of internal models being formed. Process control researchers evoke worrying images of supervisors of automated nuclear power plants trained to monitor rather than to control, and of airplane pilots whose training is excessively based on autopiloting.

Wickens and Kessel's results are important as they show how people doing similar yet different tasks on the same system develop different kinds of mental representations and different kinds of expertise. In particular, a more passive task resulted in worse learning. This is not the last time that we will run into this thought on these pages.

5.3 Teachers employ conceptual models as explanations of systems

Designing, learning about, and using a system such as a software or household application involve several different models. The users and the designers of a system have their own understandings of it, the designer has an idea of what the system's users are like, and explanatory material can be presented to users to help them interact with the system. In the field of human–computer interaction, these different models, mental and otherwise, have been called by such a variety of names that it prompted Turner and Bélanger (1996) to write a paper specifically to "escape Babel" by sorting out terminological issues regarding causal mental models. Their take is paraphrased in Table 5.1.

Table 5.1: Terms related to mental models of systems, adapted from Turner and Bélanger (1996)

Term		Definition
target system	T	A system that is designed, used, or learned about, e.g., a piece of software.
design model	$M_D(T)$	A designer's mental model of the target system.
user model	$M_D(U)$	A designer's mental model of the (stereotyped) user of the system.
system image	$I(T)$	The parts of the target system that are visible to its users, e.g., displays, controls, help files.
user's model	$M_U(T)$	A user's mental model of the target system.
conceptual model	$C(T)$	An explanation of how the target system works.

A *conceptual model* is not a mental model but an explanation of a system deliberately created by a system designer, a teacher, or someone else. Its purpose is to explain a system's structure and workings to potential users. A conceptual model may be just a simple metaphor or analogy, or a more complex explanation of the system. The aim is usually to give an accurate and consistent account of the system, but conceptual models can be incomplete and even somewhat inaccurate if the author of the model deems it appropriate for present purposes. In other words, the purpose of a conceptual model is to present some or all of the content of a design model in a pedagogically motivated way to facilitate the creation of a viable mental model. A conceptual model may be worked into the system itself, attached to it as documentation, or presented to users separately.

Conceptual models have been shown to be useful in many contexts. Schumacher and Czerwinski (1992) describe what a typical study of mental models is like: one group of learners is given procedural instructions on how to control a system ('what to do') while another group is given 'how-it-works' knowledge (a conceptual model). According to Schumacher and Czerwinski's review, the typical study concludes that those taught using a conceptual model demonstrate better performance.

Let us now take the general psychology of this chapter into the context of programming education.

5.4 The novice programmer needs to tame a 'notional machine'

The discussion in Chapter 4 highlighted the fact that writing and reading programs requires mental representations of problem-solving patterns as well as models of the program itself and the problem domain. There is another important entity that needs to be mentally represented: the computer that

executes the programs.[3] But what aspects of a computer are relevant and what can be abstracted away? Let us first read about the reasoning of Bruce-Lockhart and Norvell (2007), who describe what they are trying to teach novice programmers about.

> *The essence* [of Norman's work on mental models] *is that given a system T, a mental model of T can be defined as M[T]. Norman's work, however, was based on a very well defined T (a simple calculator). In teaching high-level (as opposed to machine language) programming it is much harder to define T. As we struggled to impart to our students that each instruction they wrote was meaningful, we had an important insight. The machine (or system) T we were programming (and which we wanted the students to understand), was not really a computer, at least in the classic, hardware, sense. Consider the following simple C code:*
>
> ```
> int x=5;
> int y = 12;
> int z;
> z = y/5 + 3.1;
> ```
>
> *In the language of programming, we say, there are four instructions to be executed. Instructions to what and to be executed by what? T of course, but T is certainly not the CPU. The first three "instructions" are actually to the compiler. We view them as requests for allocation of memory, in the stack, if they are internal declarations, in the static store if external. The fourth is a minefield. There's a truncation and two automatic type conversions. If you really want students to understand it they need to be able to interpret the expression and see the conversions, but these are normally done by the compiler. CPU operations include fetching the value for y (whether in a register or memory), carrying out the separate calculations, one in the integer arithmetic unit and one in the floating-point processor, and writing the final value back to z. We define T to be the system to which we are giving instructions. That is, T is at least partly defined by the language. In the case of C++ and Java languages, T is an abstraction combining aspects of the computer, the compiler and the memory management scheme. Our T is not nearly as "knowable" as Norman's. That does not relieve us of the responsibility of at least trying to define it. We developed* [our software tool] *the Teaching Machine to provide students with a visual representation of the T that we believe approximates the one most professional programmers program to.*

5.4.1 A notional machine is an abstraction of the computer

Benedict du Boulay was probably the first to use the term *notional machine* for "the general properties of the machine that one is learning to control" as one learns programming. A notional machine is an idealized computer "whose properties are implied by the constructs in the programming language employed" but which can also be made explicit in teaching (du Boulay et al., 1981; du Boulay, 1986).

Abstractions are formed for a purpose; the purpose of a notional machine is to explain program execution. A notional machine is a characterization of the computer in its role as executor of programs in a particular language or a set of related languages. A notional machine encompasses capabilities and behaviors of hardware and software that are abstract but sufficiently detailed, for a certain context, to explain how the computer executes programs and what the relationship of programming language commands is to such executions.

Since a notional machine is tied to a way of programming, different kinds of programming languages will have different notional machines. An object-oriented Java notional machine can be quite different from a functional Lisp notional machine[4]. Most notional machines that execute Prolog are likely to be quite different again. Similar languages may be associated with similar or even identical notional machines.

[3] If you share the extremist views of Edsger W. Dijkstra, you may disagree with this claim and argue for a computer-free CS1. See Section 14.5.3 for further discussion.

[4] A functional programming notional machine may not be very 'machine-like' at all if it is grounded in mathematics and lambda calculus. Nevertheless, I consider that such a mathematical perspective on how computer programs work when executed also falls under the term "notional machine".

Not only are there different notional machines for different languages and paradigms, but even a single language can be associated with different notional machines. After all, there is no one unique abstraction of the computer for describing the execution of programs in a language. Let us consider, for instance, the following ways of understanding the execution of Java programs.

One notional machine for single-threaded Java programs could define the computer's execution-time behavior in terms of abstract memory areas such as the call stack and the heap and control flow rules associated with program statements. The notional machine embodies ideas such as: "the computer is capable of keeping track of differently named variables, each of which can have a single value", "a frame in the call stack contains parameters and other local variables", "the computer goes through the lines of the program in order except when it encounters a statement that causes it to jump to a different line", etc. A Java notional machine at a higher level of abstraction could define the computer as a device that is capable of keeping track of objects that have been created and to pass messages between these objects as instructed by method calls in a Java program. Objects take turns at performing their defined behaviors and stop to wait for other objects whose methods they call. The computer stores the objects and makes sure each object gets its 'turn to act' when appropriate. A third Java notional machine could define the role of the computer on a relatively low level of abstraction in terms of bytecodes and the components of the Java Virtual Machine.

Notional machine: a definition

To summarize, a notional machine:

- is an idealized abstraction of computer hardware and other aspects of the runtime environment of programs;
- serves the purpose of understanding what happens during program execution;
- is associated with one or more programming paradigms or languages, and possibly with a particular programming environment;
- enables the semantics of program code written in those paradigms or languages (or subsets thereof) to be described;
- gives a particular perspective to the execution of programs, and
- correctly reflects what programs do when executed.

An obverse of the definition

I have heard computing education researchers use the expression "notional machine" to mean different things; there is also some variation in how the term is used in the literature. It is also instructive to consider what a notional machine is *not*, in my definition.

A notional machine is not a mental representation that a student has of the computer, that is, someone's notion of the machine. Students do form mental models *of* notional machines, however, as discussed below.

A notional machine is not a description or visualization of the computer, either, although descriptions and visualizations *of* a notional machine can be created (by teachers for students, for instance). Notional machines are implicitly defined by many visualizations of program execution.

Finally, a notional machine is not a general, language- and paradigm-independent abstraction of the computer. At least it is not that by definition, although notional machines can be generic enough to cover many languages, a whole programming paradigm, or even all programming languages. However, the prototypical notional machine is limited to a single language or a few similar languages. From the perspective of the typical monolingual CS1 course, the monoglot programming beginner, and this thesis, it is less generic notional machines that are usually of greater interest.

5.4.2 Students struggle to form good mental models of notional machines

In Section 5.3 above, I distinguished between a user's mental model of a target system and a conceptual model of the target system that is used for teaching purposes. In those terms, a notional machine is a target system that the CS1 student (a user) needs to construct a mental model of in order to program. A teacher, on the other hand, may wish to facilitate the creation of viable mental models of a notional machine by employing conceptual models of it.

A notional machine reflects the runtime semantics of statements. A mental model of a notional machine allows a programmer to make inferences about program behavior and to envision future changes to programs they are writing. A beginner will only have a mental model of the specific system that they are using for programming (a specific language-dependent notional machine), but as he gains in experience, he forms mental models of other notional machines and increasingly general schemas of computer behavior.

On the importance of the machine

There is plenty of agreement in the CER literature on the importance of a model of the computer for CS1 students. Du Boulay (1986) writes of the "bizarre" ideas that students have of how the computer executes programs, and identifies the notional machine as one of the main areas of difficulty that the programming novice needs to come to grips with. According to Cañas et al. (1994), "it is widely accepted that programming requires having access to some sort of "mental model" of the system". Perkins et al. (1990) "attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer". Smith and Webb (1995a) state that novices' difficulties in developing and debugging their programs stem from the fact that "their mental model of how the computer works is inadequate". Ben-Ari (2001a) concludes from the literature that "intuitive models of computers are doomed to be non-viable" and that novices' lack of an effective model of a computer can be a serious obstacle to learning about computing. And so on.

The lack of a viable model of the computer can lead to misconceptions, difficulties with understanding program state, and problems in knowledge transfer. I will comment on each of these topics in turn.

On misconceptions and hidden processes

> *A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes.* (du Boulay, 1986)

A computer program has two forms: static and dynamic. The static aspect of a program is visible in code, but the dynamic aspect is usually implicit. The hidden nature of program dynamics has been linked to the multitude of misconceptions about programming concepts that students have.

In Section 3.4, I gave a lengthy list of studies that have uncovered misconceptions of programming concepts. Many of the inadequate understandings and difficulties discovered in those studies can be explained by the lack of a viable mental model of the notional machine that one is learning to control. Sleeman et al. (1986) concluded as much after reporting numerous misconceptions about Pascal programs: "even after a full semester of Pascal, students' knowledge of the conceptual machine underlying Pascal can be very fuzzy". With reference to the literature on misconceptions, Sajaniemi and Kuittinen (2008) likewise attribute student difficulties with basic concepts to a lack of understanding of a notional machine. Kaczmarczyk et al. (2010) identify, from their empirical data, "the relationship between language elements and underlying memory usage" as a major theme in students' misconceptions.

The list of misconceptions in Appendix A provides many examples of the kind of "hidden, internal changes" within the notional machine that du Boulay (1986) noted as being problematic for students. Consider for instance the notion that the object assignment `a = b` (in Java) copies the values of an object's instance attributes to another object. Overcoming this misunderstanding requires the concept of a reference to an object, which is something that is not apparent in code. Many misconceptions, if not most of them, have to do with aspects that are not readily visible, but hidden within the execution-time world of the computer: references, objects, automatic updates to loop control variables, and so forth.

Some generic misconceptions may lie behind many of the other more specific misconceptions. A few such generic misconceptions are listed at the beginning of Appendix A – for instance, it is thought that

the computer can carry out deductions regarding what the programmer intends to do. These generic misconceptions concern the capabilities of the computer and/or the execution-time behavior of programs. In other words, they indicate problems with students' mental models of notional machines.

Subtle, 'hidden' aspects of programs also top some polls on difficult CS1 topics. Milne and Rowe (2002) surveyed students' and tutors' opinions of the difficulty of programming concepts. They conclude that the most difficult concepts, such as pointers, have to do with the execution-time use of memory, and that "these concepts are only hard because of the student's inability to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution." A Delphi survey of computing educators identified references, pointers, and an overall memory model as some of the most difficult topics in introductory programming (Goldman et al., 2008). The students from various educational institutions that were surveyed by Lahtinen et al. (2005) found pointers and recursion to be the most difficult topics.

It is not just what the computer does behind the scenes that needs to be understood. The novice must also realize what the notional machine does *not* do, unless specifically instructed by the programmer. People do not naturally describe processes in the way programmers need to – the equivalents of `else` clauses, for instance, are conspicuous by their absence in non-programmers' process descriptions, as people tend to forget about alternative branches and may consider them too 'obvious' to merit consideration (Miller, 1981; Pane et al., 2001). The novice needs to learn what the notional machine does for them on the one hand, and what their own responsibility as a programmer is on the other.

Difficulties with program state

How the computer keeps track of program state is one of the central aspects of most notional machines. However, execution-time state is generally not explicit in program code. This is a source of confusion for novices, who "sometimes [forget] that each instruction operates in the environment created by the previous instructions" (du Boulay, 1986). Concrete examples of student difficulties can be found in the work of Sajaniemi et al. (2008), who studied student-created visualizations of the states of object-oriented programs. Their results showcase the variety of misconceived ways in which students envision state.

A finding from program comprehension experiments (Section 4.6) that has received relatively little attention is the difficulty of comprehending state-related aspects of programs. Corritore and Wiedenbeck (1991) compared novices' ability to answer questions about elementary operations, control flow, data flow, program function, and program state after reading a small program. They found that questions about state had a dramatically and unexpectedly high error rate. A comparison of upper and lower quartile novices showed that they differed particularly in their ability to answer questions about program state (although both had the most difficulty with state questions compared to other kinds). According to another study reported in the same paper, the difference between upper and lower quartile novices' ability to answer questions about state was even more pronounced when dealing with longer programs. Corritore and Wiedenbeck contrast their findings on novices with Pennington's (1987b) comparable study on experts, which showed no such highly elevated error rate on state questions. They conclude that understanding state is an area where novices differ from experts.

Notional machines and transfer

> *Programming should not be taught as a copy-paste art that only incidentally results in a correctly functioning program, but as a clearly defined activity that deals with unambiguous constructs.* (Sajaniemi and Kuittinen, 2008)

The importance of the notional machine is emphasized in situations where knowledge needs to be transferred to new contexts or where creative solutions, rather than familiar templates, are needed. Furthermore, research shows that pedagogy can have a significant impact on how well students can transfer their knowledge.

Kessler and Anderson (1986) conducted a sequence of studies on novice learners' iteration and recursion. One group of novices learned recursive programming first, followed by iterative programming, and another group were introduced to the topics in the reverse order. Neither group was explicitly

taught about a notional machine. In this study, the students who started with iteration initially formed better mental models of control flow, which they were later able to transfer to the novel context of recursive programming. In contrast, the students who started with recursive programming tended towards a template-based programming style in which they tried to match the surface features of problems to the surface features of known program examples. Consequently, the recursion-first group managed to solve certain kinds of recursive problems, but failed to transfer what they knew to iterative programming, becoming "overwhelmed by the surface differences between recursion and iteration". A protocol analysis suggested that the recursion-first group did not construct a model of the implicit principles of control flow underlying the example programs they saw – in other words, they had failed to understand the notional machine. In terms of Schumacher's theory of mental model formation (Section 5.2 above), it might be said that the students in Kessler and Anderson's studies who started with iteration soon reached the experiential stage with respect to possessing a mental model of a notional machine that supports repetition, whereas those who started with recursion struggled to get past the pretheoretical stage. A similar study by Wiedenbeck (1989) produced results compatible with Kessler and Anderson's.

For present purposes, what is most significant about these studies is that while even novice programmers manage to solve certain kinds of problems without being taught about a notional machine, a viable mental model of a notional machine is needed to understand programs on a deeper level and to transfer that understanding to new contexts for which ready-made templates are not available. No machine model is needed if the goal is to produce learners who will not use the programming language creatively, but if and when the goal is to produce creative problem solvers, then learning about the machine early on is "quite useful", as Mayer (1981) mildly puts it.

The birth of naïve models

Unfortunately, novices' mental models are often based on mere guesses.

Mental model theory tells us that people rely heavily on analogies based on surface features when forming mental models of new systems they encounter (Section 5.2). There is evidence of this in programming education as well. As du Boulay points out, a notional machine's properties are implicit in the constructs of the corresponding programming language (which is, in terms of Table 5.1, a key aspect of the system image of the notional machine). Indeed, program code is a fertile basis for constructing a mental model as "novices make inferences about the notional machine from the names of the instructions" (du Boulay et al., 1981). Many programming languages, on the surface, resemble natural language and the language of mathematics. Misconceptions (Section 3.4 and Appendix A) are brought about by unsuccessful analogies with these realms, such as when students conclude that the Java statement `a = b + 1;` defines a mathematical equation.

Mental model theory further explains that novices may have several contradictory mental models of a notional machine that they use to deal with different scenarios, whereas an expert's mental model is more generic and stable. Assignment statements with integer variables might be explained with one mental model, for example, and assignments using record types with an entirely different one.

The impact of environments

Programming paradigms and programming environments can make a difference to learning about program dynamics. Some existing environments and perhaps especially the programming environments of the future blur the line between development time and program runtime – consider, for instance, the Smalltalk environment, the BlueJ IDE (Kölling, 2008), the DISCOVER tutor (Ramadhan et al., 2001), and the recent work of Victor (2012). Such developments bring many exciting benefits to both novice and expert programmers. They may also introduce some pedagogical challenges. For instance, as Ragonis and Ben-Ari (2005a,b) studied high-school students learning object-oriented programming, they "became aware of serious learning difficulties on program dynamics" as "students find it hard to create a general picture of the execution of a program that solves a certain problem". They suggest that object-oriented modeling and pedagogical tools that involve direct manipulation of objects while authoring a program (such as BlueJ) may exacerbate this difficulty.

Therefore: teach early and teach long

Studies of mental models suggest that people cling to initial models, so that fixing a 'broken' model can be more work than constructing a viable model in the first place. This is one reason why it is important to help students form a workable mental model of a notional machine early on in their programming studies. Another concern is that a seriously flawed model of a notional machine may work for explaining the behavior of some program examples, even though it fails generally, which further deepens the learner's belief in their present understanding. While students obviously do not come in as blank slates, no matter what teachers do, getting in as early as possible seems a good idea.

Starting early does not mean ending early, either. Programming teachers often expect students to be able to switch between programming languages (and notional machines!) fairly early on in programming curricula, commonly after a first programming course. While transfer on the basis of superficial features is commonplace, transfer to even similar notional machines that look dissimilar on the surface (i.e., in program code) requires a deeply ingrained mental model of the original notional machine. As forming such models tends to require a substantial amount of experience, teachers need to make sure that students get plentiful practice with the original notional machine before expecting them to transfer to other languages and paradigms.

Learning about a notional machine can draw on a conceptual model that makes explicit the way in which the machine works and underlines how a programming language differs from natural language and familiar mathematics. A conceptual model of a notional machine can take the form of a drawn visualization, a verbal or textual description, or a software application that visualizes the notional machine. It makes explicit the hidden effects of commands in program code. In these terms, what Bruce-Lockhart and Norvell (2007) describe (p. 59 above) as their target system T is a notional machine for C programming, and the visualization in the Teaching Machine software serves as a conceptual model for teaching about this notional machine.

I will discuss conceptual models of notional machines in more detail in Part III. Visual program simulation, which is introduced in Part IV, is a way of engaging students in interactions with a conceptual model of a notional machine.

The notional machine is also involved in the activity of program tracing.

5.5 Programmers need to trace programs

Tracing a program means analyzing its execution to determine what operations occur and how its state changes. The human act of tracing a program can be viewed as a form of mental simulation (cf. Section 5.1.3). Tracing is a key programming skill that expert programmers routinely use during both design and comprehension tasks (Adelson and Soloway, 1985; Soloway, 1986). As discussed in Section 3.3, developing the ability to trace programs is linked – at least to some extent – to the development of program comprehension and program writing skills.

5.5.1 Novices especially need concrete tracing

Concrete vs. symbolic tracing

Détienne and Soloway (1990) distinguish between two different techniques that experienced programmers use when trying to comprehend a program: symbolic and concrete tracing (also known as symbolic and concrete simulation, respectively).

Symbolic tracing means using generic values while tracing a program's execution, e.g., "it loops ten times, reads a number, compares the number to maximum and sets Max to Num if something is true" (Détienne and Soloway, 1990). This was the default strategy used by the experienced programmers studied by Détienne and Soloway when they first encountered a new program. They used it whenever possible, that is, as long as they thought that the program matched their existing plan schemas and no problems were evident. *Concrete tracing*, on the other hand, uses specific values: "the number should be five, so the average should be five, I put five in Sum, add 1, Num is not 99999 [the sentinel value], enter 99999 to get out, Sum is now five, Count is one." (ibid.) Experienced programmers used concrete tracing, in

Détienne and Soloway's parlance, "to evaluate the external coherence between plans, i.e., to check for unforeseen interactions". In other words, concrete tracing was useful for studying programs which merged familiar plans in an unfamiliar way. For experts, concrete tracing is a "last resort in cases that are complex, hard to understand, or when the program does not work the way that symbolic tracing suggests" (Vainio and Sajaniemi, 2007).

Concrete tracing and debugging

Concrete tracing is a vital skill for any programmer. It is especially useful in the fight against bugs: the programmer may know what the various code constructs do in isolation, but not what happens when they are combined in the buggy way they currently are, and need to "check for unforeseen interactions".

Concrete tracing is even more important for novices. Vainio and Sajaniemi (2007; Vainio, 2006) argue that the novice programmer needs to trace code carefully in order to understand the causal relationships between statements, and cannot use symbolic tracing as the default strategy. The novice lacks plan schemas that provide solutions to common problems, which hinders symbolic tracing and can force the novice to work at the concrete level. Before they can raise the level of abstraction during tracing, novices need to automate – through experience and practice – the processing of syntactic and semantic details.

Some important skills are pedagogically unproblematic. Unfortunately, as we have already seen in Section 3.3, there is plenty of evidence that many CS1 students fail to learn how to trace programs. Moreover, Perkins et al. (1986) found that many novices do not even try to trace the programs they write, even when they need to in order to progress. In their study, "students seldom tracked their programs without prompting". Failure to trace one's programs, Perkins et al. argue, may be due to reasons such as a failure to realize the importance of tracing, a lack of belief in one's tracing ability, a lack of understanding of the programming language, or a focus on program output rather than on what goes on inside.

5.5.2 Tracing programs means running a mental model

One of Perlis's (1982) famous epigrams states that "to understand a program you must become both the machine and the program". Similar sentiments have been expressed in the psychological research literature.

One might view program tracing as running a mental model of a program with some input, while also running a separate mental model of a notional machine for which the program itself serves as input. However, when we run a program, "the computer effectively becomes the mechanism described by the program" (du Boulay, 1986). When we speak of program execution, we use expressions such as "the program does X", as well as "the computer does X", to mean effectively the same thing. As noted earlier in the chapter, the borders of mental models are often vague. Even though the models of the machine and program can be viewed analytically as separate, they may be inextricably entwined during mental tracing and are not necessarily distinct in the programmer's memory. For present purposes, it is convenient to speak of mental tracing as the 'running' of a single mental model that encompasses both the notional machine and the program that is being traced.

Two challenges to the successful running of a mental model during tracing are the difficulty of keeping track of program state in working memory, and the robustness of the mental model.

Status representations

Tracing a program requires keeping track of the state the execution of the program. Such a description of state, which Perkins et al. (1986) call a *status representation*, must be dynamic, changing as execution proceeds. A status representation consists of the elements of (one's mental model of) the notional machine used: variables, objects, references, function activations, etc.

As discussed in Section 5.1.3, mental tracing is taxing for working memory, and success is predicated on carefully choosing a level of abstraction and the 'moving parts'. People tend to run their mental models qualitatively rather than with specific values, and correspondingly prefer symbolic to concrete tracing. However, as noted above, programming tasks like debugging call for concrete tracing. A status

representation of a complex program involves an amount of information that often exceeds the capacity of working memory, which is why we use external aids such as scraps of paper and debugging software.

Novices in particular need concrete tracing often, but are not experienced at selecting the right 'moving parts' to keep track of in the status representation, causing them to fail as a result of excessive cognitive load (see, e.g., Vainio and Sajaniemi, 2007; Fitzgerald et al., 2008). Vainio and Sajaniemi (2007, p. 239) discuss how failure to abstract can lead to overwhelming cognitive load: "Tracing a sorting algorithm without [the abstraction of a swap operation] may result in too many objects taking part in the simulation, which may cause the simulation to exceed the limits of working memory and become error-prone or too slow to be practical.[5] Vainio and Sajaniemi discovered a novice strategy (commonness unknown) for concrete tracing which is motivated by its low load on working memory. This strategy, which they call "single-value tracing" limits the number of non-trivial variable values in one's mental status representation to one.[6] That is, only a single unnamed 'slot' is used to store the value of whichever variable was most recently non-trivially assigned to. Single-value tracing does not work in the general case, but the novice may not even realize this as "it works fine with small programs that are typical to elementary programming courses".

Despite problems with mental status representations, novices tend not to use external aids as much as they need to. Lister et al. (2004) report that the most common type of 'doodle' on paper used by students during a tracing task was no doodle at all. Thomas et al. (2004) experimented with object diagrams that describe runtime state, essentially an external visual aid for object-oriented status representation. They found that novice programmers were not helped by object diagrams that they were given, and were distinctly lacking in eagerness to draw diagrams themselves. The results of Vainio and Sajaniemi (2007) suggest that there are two kinds of difficulties: novices not only struggle to produce state diagrams, but also fail to make use of such diagrams when they have produced them. Moreover, anecdotal evidence from CS1 courses suggests that students do not voluntarily use debuggers – another form of external status representation – as much as their teachers would like them to (this is my own experience; see also, e.g., Isohanni and Knobelsdorf, 2010).

Robust models needed

According to de Kleer and Brown's theory (Section 5.1.3 above), a mental model of a causal system needs to be robust if it is to be transferred to novel component configurations. If de Kleer and Brown's theory holds, and assuming that computer programs are causal mechanisms of the sort that their theory applies to, then for students to transfer their understanding of one program to other programs, they need a robust model of the original program. The components of a program are the programming constructs used in the program code and the corresponding mechanisms within the notional machine. To stay with de Kleer and Brown's theory, models of these program components ideally follow certain "esthetic principles" of robustness. De Kleer and Brown also contended that a model must be robust to be usable in unexpected contexts where the behavior of the system does not follow one's prediction. A robust mental model of both the program and the associated notional machine would therefore be needed to debug programs.

Vainio (2006; Vainio and Sajaniemi, 2007) applied de Kleer and Brown's theory to programming. They focused in particular on the no-function-in-structure principle and the locality principle that supports it (Section 5.1.3). Consider the following code fragment.

```
for (i = 0; i < 10; i++) {
    System.out.println(i);
}
```

[5] When the program is sufficiently complex, experts will also be overwhelmed by cognitive load. Parnas (1985) criticized tracing as a software development tool: "As we continue in our attempt to "think like a computer," the amount we have to remember grows and grows [along with program complexity]. The simple rules defining how we got to certain points in a program become more complex as we branch there from other points. [...] Eventually, we make an error." However, as Soloway (1986) pointed out in response, tracing is the best way invented so far for understanding the dynamic aspect of programs, and therefore should be learned by students. Programmers today have access to ever better software tools that support mental tracing and sometimes even eliminate the need for it, but Soloway's observation still appears to hold largely true.

[6] 'Trivial variable values' are those that can be seen directly from the program code, such as literals assigned to variables.

Figure 5.1: Robust models transferred into an unexpected context (Rubens, ca. 1623).

A reasonable description of this `for` loop is: "First, the variable – `i` in this case – is set to zero. Then the loop iterates over all the values of `i` from 0 to 9 and prints them out." A novice programmer may form a mental model of this program that matches this description and is viable when it comes to dealing with this specific program, but is not robust. For instance, some of the students in Vainio's study had developed an understanding according to which *whichever variable* is used within the body of a `for` loop is *always* set to zero at the start of the loop. Such an understanding is clearly not generally viable.

A problem with an understanding such as the above is that the student has formed a component model of the `for` statement (within their model of the program) that does not meet the principles set out by de Kleer and Brown. The understanding violates the no-function-in-structure principle, as it mixes up the definition of a `for` statement (its structure) with what the `for` statement is used for in a particular context (its function, which here includes assigning zero to a variable). A related violation is that of the locality principle. The component model of the `for` construct is not independent of the specifics of other constructs; it is dependent on the idea that somewhere within the body of the loop there is at least one other statement that makes use of a variable that affects what the `for` statement does.

Vainio and Sajaniemi (2007) describe violations of the no-function-in-structure principle as common, attributing this in part to the tendency in CS1 courses to associate each type of problem with only a single kind of programming construct, and each programming construct with a single kind of problem. Their work emphasizes the need for novices to separate what a construct is and what it is typically used for.

To summarize this section, tracing a program requires the ability to mentally simulate programs running in a notional machine. Novices especially need concrete tracing at a low level of abstraction to make sense of programs, but often lack the ability or even the inclination to trace programs. Ideally, mental simulations are based on robust, generalizable mental models of programs, and rely on abstraction and external status representations if and when the limits of working memory are met.

5.6 Where is the big problem – misconceptions, schemas, tracing, or the notional machine?

Let us reflect on what this and the previous chapters have told us about learning to program. My review of learning programming has so far centered around five challenges that face the novice programmer.

1. Creating programs imposes a great cognitive load on novice programmers.
2. Programmers need plan schemas which represent generic solutions to common problems, but novices have few.
3. Novice programmers have misconceptions about basic programming concepts, which give them trouble when reading and writing programs.
4. Many creative and unexpected programming tasks require mental tracing of programs, something that novices are not always capable of.
5. Novices need to form a viable mental model of a notional machine to be able to understand program execution.

Some pertinent and difficult questions are as follows: Which of these challenges are the most important? Is there a particular bottleneck for learning? How do the challenges depend on each other?

Different researchers have given different answers to these questions.

A legion of misconceptions

Research on misconceptions has a long history which demonstrates that non-viable understandings of programming concepts have been considered an important and pedagogically fertile area of computing education research by numerous authors. Clancy enthuses:

> *To me, this* [research on misconceptions and mistaken attitudes] *is the most interesting and, for my teaching, the most relevant education research. It helps me interpret wrong answers – which I see a lot of! – and also guides me toward activities that address student problems.* (Clancy et al., 2001, p. 330)

As a teacher, it is easy to agree with Clancy that knowing about the kinds of mistaken understandings that students have is very useful in practice, and can help in the design of better programming courses and communicating better with students. On the other hand, some of my colleagues have called research on misconceptions passé or pointless, as there are presumably an infinite number of incorrect ways to understand a concept and charting out all these ways is a futile effort.[7] While I certainly do not consider such research pointless, it is true that the multitude of misconceptions is an issue. If we manage, through research, to get at the general causes underlying some of the more specific misconceptions, we gain better insight into learning programming and can perhaps swat out entire families of novice bugs in one stroke.

Some phenomenographers have argued that rather than misconceptions, we should be concerned with the different correct but partial ways in which people understand programming concepts. Research à la Eckerdal and Thuné (2005) can point out the educationally critical aspects of programming concepts, which – according to the constitutionalist perspective on learning (Chapter 7) – are limited in number. A different approach that seeks to explain why certain kinds of understandings are not viable is Vainio's application of de Kleer and Brown's 'esthetic principles' to programming (Section 5.5 above). Soloway and Bonar's 'bug generators' and Pea's 'superbug' of the anthropomorphic computer (Section 3.4) also aim to get at the general reasons behind various specific misconceptions.

My last-not-least example of a principled way to address numerous misconceptions at once was already discussed in Section 5.4: the notional machine. To recapitulate, it seems that many non-viable understandings can be explained by the root cause of novice programmers not understanding program execution and the role the computer plays in it. While little empirical work exists that has investigated whether or how particular misconceptions can be helped by learning about a notional machine, there is plenty of general evidence and agreement in the literature that learning how the machine works is important.

Soloway & Spohrer: schemas > misconceptions + syntax

A line of thinking influentially argued for by Spohrer and Soloway (1986a,b) is that syntax and misconceptions about language constructs are not as significant a problem for learning programming as the lack of plan schemas that enable novices to solve common problems. Spohrer et al. (1985) attribute many novice mistakes to an inability to successfully merge plans. Winslow (1996) claims that "study after study has shown that students have no trouble generating syntactically valid statements once they understand what is needed. The difficulty is knowing where and how to combine statements to generate the desired result."

Prerequisites of high-level schemas

What is required to use programming schemas to solve problems? Misconception-free understandings of syntax and semantics, and some skill at tracing, at least.

When emphasizing the importance of plan schemas, Spohrer and Soloway referred especially to higher-level plan schemas (e.g., the find-average schema) rather than low-level ones (e.g., the assign-to-variable schema). As discussed in Section 4.4.1, high-level schemas build on low-level ones. To be useful, these low-level problem-solving schemas must contain viable, non-fragile understandings of the semantics of programming constructs and the associated notional machine. Misconceptions about fundamental programming concepts lead the novice to ignore pertinent aspects of a situation and miss links between examples, reducing germane cognitive load and hindering the formation of higher-level schemas. Misconceptions can also lead to the formation of 'buggy schemas' at higher levels.

As discussed in Section 3.3, many novices lack tracing skills, which are important for many debugging tasks and bug-free problem solving using plan schemas. One study explicitly concludes that their findings

[7] Personal communication.

are "somewhat contrary to the classic work of Spohrer and Soloway": students are sometimes able to create buggy template-based (schema-based) code, but do not necessarily understand the code that they themselves produce, and cannot trace its execution to fix the bugs (Thomas et al., 2004).

Overall, recent research does not disagree with Spohrer and Soloway's identification of problem-solving schemas as important, but does emphasize that lower-level issues are also worthy of attention.

Garner, Haden, and Robins categorized the problems that CS1 students needed help with in class and examined the resulting distributions (Garner et al., 2005; Robins et al., 2006). Many of the problems had to do with program design, but many also involved various construct-related issues. Trivial mechanical problems (e.g., missing semicolons) were also common. The distribution pattern was similar across course offerings; high-, medium-, and low-achieving students also all exhibited similar patterns. Fitzgerald et al. (2008) report that novice programmers with 15 to 20 weeks of programming experience behind them had more trouble finding non-construct-related bugs than construct-related ones. Ko and Myers (2005) presented a framework for analyzing the causes of software errors; applying it, they found that many bugs were rooted in cognitive difficulties with particular language constructs, although other causes of difficulty were common as well. Denny et al. (2011) demonstrated that novices have great trouble in producing even small programs that are syntactically correct. McCauley et al. (2008) point out that construct-related bugs feature even in a revered pedant's classification of the bugs in TeX (Knuth, 1989).

Drawing on the BRACElet studies (Chapter 3), Lister (2011b,c) has recently emphasized that although struggles with syntax initially contribute to students' cognitive load – and are important to address – the great cognitive load inherent in program writing remains a major concern even when those earliest problems have been overcome. Kranch (2011) studied novice programmers taught using three different topic sequencings, including one that started with higher-level schemas before moving to their elements, and one that did the opposite. He found that irrespective of ordering, the higher-level schemas were always the most difficult topic (both in terms of achievement and student-given difficulty ratings), a finding that he attributes to higher intrinsic cognitive load. Kranch argues that novices should be given ample opportunity to practice lower-level fundamentals before they are taught higher-level schemas.

The studies from the 1980s that pointed to the relatively minor importance of misconceptions used simple imperative programs. It is not obvious to what extent the findings are generalizable to object-oriented programming, which has brought a slew of new concepts and misconceptions into many CS1 courses. Many OOP misconceptions are about the fundamental nature of pivotal concepts such as object and class (see Appendix A).

Conclusion: they all matter

My conclusion from the literature is that all five challenges – cognitive load, plan schemas, misconceptions, tracing skills, and notional machines – are significant. Furthermore, none of these challenges is independent of the others. Of particular interest to the present work is the fact that the other four challenges appear to be affected by the fifth: the novice's ability to understand the notional machine. Understanding the role of the machine in program execution:

- prevents and corrects numerous misconceptions;
- serves as a basis for the formation of low-level schemas which in turn form the basis for the successful formation and application of higher-level ones;
- thereby indirectly reduces cognitive load as one is able to rely on increasingly complex problem-solving schemas, and
- is key to the skill of tracing programs, which is useful in both program authoring and comprehension and in finding bugs whether they be misconception-related or the results of high-level schema failures.

The notional machine is not a singular bottleneck responsible for students' struggles, but it does appear to be one of several main sources of difficulty.

5.7 The internal cognition of individual minds is merely one perspective on learning

Cognitive psychology has made useful, empirically warranted contributions to the study of learning, but it has also come in for criticism from various quarters. Researchers from a number of overlapping camps – phenomenologists and phenomenographers, proponents of situated learning, qualitative researchers, philosophers, and neo-behaviorists, as well as cognitive psychologists themselves – have pointed out the limitations and possible fundamental problems of cognitive science. Before we turn to other traditions of educational research in the following chapters, let us consider some of the complaints concerning the cognitive tradition, or *cognitivism* as it is sometimes dubbed.[8]

Controversy 1: internal representations

Cognitivism tends to see reasoning in terms of the manipulation of symbols. A famous argument against internal mental representations is the problem of the *homunculus* (see, e.g., Marton and Booth, 1997, pp. 9–10): it is claimed that to operate on an internal representation of the world one needs something other than the representation itself – a so-called homunculus, or a "little human in the head". Following the same logic, the homunculus represents the representation internally, requiring another homunculus within it. This leads to an infinite regress. Many solutions to the problem have been proposed. Marton and Booth's self-statedly 'non-psychological' solution sidesteps the little human by choosing to research not mental representations at all but the relationships of people to phenomena as embodied in dialogue, actions, and artifacts. I will explain their phenomenographic approach further in Chapter 7.

Another way to answer the conundrum is to assume that the cognitive apparatus that operates on internal representations is innately capable of manipulating entities of the general form that internal representations have. That is, operating on an internal representation does not require a representation of the representation but only the ability to operate on representations of that general type. The traditional cognitivist way to phrase such an answer is to liken the human cognitive system to a computational device which contains both mental representations (data) and procedures that manipulate the representations (see, e.g., Markman, 1999, p. 13). This answer brings us to our second controversial theme.

Controversy 2: mind as machine

Cognitivism is associated with the idea that the human cognitive system resembles a general-purpose computer. Within this apparatus, information is stored within memory and processed algorithmically. Because of this analogy, cognitive science has been criticized for "vastly oversimplifying both the human mind and the human brain" (Westbrook, 2006), "for imposing severe constraints on potential descriptions and explanatory models" (Marton and Booth, 1997), and for committing so heavily to the analogy of a computer as to ignore consciousness and intuition (Searle, 1992; Dreyfus, 1992; Klein, 1999).

Perhaps the most fruitful way to address this point is to accept that the information-processing view can serve – as it has – to provide insights into the human mind, but that – like all analogies – it has its limits. Other perspectives are also needed.

Controversy 3: individual, context-free knowledge

Most cognitivists view reasoning primarily as a mental operation performed by individuals. Exploring the social situatedness of learning – more on which in the next chapter – has not traditionally placed especially high in the cognitivist agenda. According to proponents of some social theories of learning, cognitivists tend to ignore or pay too little attention to the relationship between individual thought and social context, and the way knowledge is shaped and problems are solved socially rather than individually. Researchers into situated cognition (cognition embedded in physical/social/cultural contexts) have been critical of the dominant role of internal memory in much of cognitivist research, arguing instead for a focus on situated perception and knowledge that is created on the fly, in activity, rather than retrieved from

[8] It is my impression that the term "cognitivism" is used more often by detractors of cognitivism than by its proponents. I do not use the word in a pejorative sense.

memory storage (see, e.g., Greeno, 1994; Anderson et al., 2000a, and references therein). An example of a moderate position is that of Anderson et al. (2000a), who seek to mediate the conflict. They sensibly point out that the individual and the social perspectives are not incompatible but provide two complementary views on the study of learning and thinking.

Controversy 4: content-independent cognition

A related issue is that many cognitivists seek to discover generally applicable principles and forms of representation (e.g., schemas) that explain activities in any field of knowledge and in any context. This universalist goal has been criticized by phenomenologists (e.g., Dreyfus, 1992) and phenomenographers (e.g., Marton and Booth, 1997), who prefer to investigate specific phenomena and people's relationships to specific phenomena. Such critics see generic cognitive psychology as flawed because the specific content that is processed is of fundamental importance. In seeking to differentiate their work from cognitive psychology, Marton and Booth (1997) argue that "the general can only be revealed through the specific" and that "ideas and principles need to be developed anew in specific contexts and contents of learning and teaching". This criticism does not acknowledge the fact that results in cognitive psychology, too, can arise from specific contexts, and sometimes apply only to a specific domain. For example, some of the work presented in this chapter and the previous one has "developed anew" general ideas from psychology, using programming education for context and content (e.g., programming plans and the roles of variables). Nevertheless, Marton and Booth's criticism serves as a reminder to moderate the eagerness for wide generalization that typifies cognitivist work.

Controversy 5: accessibility

Cognitivism is based on the notion that mental entities are accessible to researchers. Behaviorists – and others – have accused cognitivism of overestimating researchers' ability to elicit people's mental representations and analyze them objectively. A related claim is that in seeking to explain thought, cognitivists are as likely to invent one internal mechanism as another. To Uttal (2000, p. 12), "it seems that many cognitive psychologists are not deeply enough concerned with the fundamental issue of accessibility. Usually, the issue is finessed and ignored." He is concerned that the inaccessibility of mental processes leads to poor experiments with "embarrassingly transient" results that tend to be soon contradicted.

> *As I studied and reviewed the "high level" literature, I came to a rather surprising general conclusion. The reliability, durability, and presumably the validity of the data from the sample of experiments with which I was concerned seemed to evaporate. Data, as well as conclusions, seemed to last only for a few issues of the journal in which they had been published before some criticism of it emerged. Or, when the experiment was repeated, slight differences in the design produced qualitatively distinct, not just marginally different quantitative results. My summary statement on this issue still expresses my conviction that the high level, cognitive aspects of perception and other aspects of mentation are far less accessible than many mentalists would accept simply because the very database is so fragile.* (Uttal, 2000, p. 77)

The results of Uttal's review are worrying and must be taken seriously. Still, there also certainly exists evidence within cognitive psychology of the convergence of results. In some cases, mixed results may be later clarified by an improved, unifying theory; we have seen examples of this in Section 4.4.2, in which Rist's theory of schema formation helped make sense of earlier mixed results, and in Section 4.5, in which the discovery of the expertise reversal effect helped make sense of confusing results concerning several other effects of cognitive load.[9]

Controversy 6: dualism

Cognitivism has been tagged with the "original sin of dualism" by critics from different backgrounds (e.g., Uttal, 2004, 2000; Uljens, 1996; Marton, 1993; Marton and Booth, 1997). A dualist view of ontology

[9] Problems of the sort Uttal mentions nevertheless continue to concern cognitive load theorists, as discussed by Moreno (2006).

maintains that physical and mental objects are fundamentally disparate and 'inhabit different worlds' or consist of different substances. Dualism is commonly criticized, among other things, for failing to explain how it is possible for mind and matter to interact. Many cognitivists, however, denounce dualism and prefer a monist, materialist position in which the mental is viewed as reducible to physical phenomena. Another alternative to traditional dualism is *property dualism*, which draws a dichotomy not between substances of mind and matter but rather between objective physical and subjective mental phenomena that arise out of a single physical matter but are fundamentally distinct in character from each other. These philosophical positions, and others, have been reviewed, e.g., by Searle (1992, 2002), who himself promotes *biological naturalism*, the "fairly simple and obvious" solution that mental phenomena are physical but additionally have higher-level, emergent characteristics that are irreducible to physics.

Mind–matter dualism has been a source of bother and employment for philosophers for a long time, without a generally accepted solution in sight.

Controversies 7 and 8: quantitative "positivist" research

Finally, since cognitivism has traditionally been characterized by quantitative, experimental, "positivist"[10] research, it has come in for criticism from various quarters that are opposed to those paradigms. "Positivism" and quantitative research have been criticized on ontological, epistemological, and practical grounds by qualitative researchers (e.g. Lincoln and Guba, 1985; Patton, 2002) and others. Claimed weaknesses include:

- a reliance on a correspondence theory of truth, that is, the idea that facts correspond to a one real world,
- a naïve belief in objectivity; ignoring the context-, subject- and theory-dependence of facts,
- equating the natural and social sciences,
- the belief that (all) science is transcultural,
- losing richness of data and validity of research through the reductionist attempt to operationalize complex phenomena into simple variables,
- being good only for hypothesis testing rather than exploration,
- a focus limited to laws, causes, and predictions as opposed to multiple interpretations,
- ignoring the societal embeddedness of phenomena,
- invalid results that ignore the humanness of research subjects and researchers,
- the idea that inquiry can be value-free, or at least should be as value-free as possible,
- vain attempts at wide or universal generalization, and
- an obsession with the reproducibility of experiments.

Obviously, the extent to which these criticisms are applicable to particular cognitivists and studies within cognitive psychology varies. Controlled laboratory experiments in the quantitative tradition do have a credible track record of success in motivating educational reform (see, e.g., references in Atkinson et al., 2000). They are also hardly the only form of cognitivist research. Many of the CER studies in the cognitivist tradition that I have reviewed in this and the previous chapter are actually qualitative in character (e.g., much of the work on identifying programming misconceptions and plans).

A detailed review of this long and complicated debate is well beyond the scope of this thesis. For the present, I will only note that cognitivist research and its competing paradigms have different strengths and weaknesses. In my own empirical work for this thesis in Part V, I take a pragmatist approach that mixes paradigms.

[10]The word "positivism" has been defined in many different ways, and is often not very well defined at all when employed as a bludgeon (Phillips, 2004). I use it here, in quotes, to refer loosely to various criticized views that have been assigned this label. Typically, "positivists" are associated with the view that all scientific knowledge must be founded on value-free empirical testing of hypotheses.

Chapter 6

Constructivists Say: Knowledge is Constructed in Context

The central tenet of the educational paradigm known as *constructivism* is that people actively construct knowledge rather than passively receive and store ready-made knowledge. Knowledge is not taken in as is from an external world, and is not a copy of what a textbook or teacher said. Instead, knowledge is unique to the person or group that constructed it – constructivists differ among themselves as to whether individual minds or social groups (or both) are the constructing agents. From these premises, constructivist thinkers have derived pedagogical recommendations which tend to promote active, learner-centered education.

Proponents of constructivism often claim a positive, even revolutionary impact on education; skeptics either point at perceived flaws in constructivist reasoning or dismiss constructivism as not so much a revolution as a rephrasing of prevalent wisdoms. These issues notwithstanding, many critics, too, agree that the currently extremely influential constructivist movement has done good by bringing epistemological issues to the forefront of educational discussions, by advancing the increasingly widespread recognition of the social aspects of learning and the importance of learners' prior knowledge, and by emphasizing active learning (Phillips, 1995, 2000).

To get an overall feel, let us begin with a list of selected constructivist claims (my phrasings based on von Glasersfeld, 1982; Phillips, 1995, 2000; Steffe and Gale, 1995; Greening, 1999; Ben-Ari, 2001a; Larochelle et al., 1998; Rasmussen, 1998; Anderson et al., 2000b; Patton, 2002; Kirschner et al., 2006; Tobias and Duffy, 2009). These claims range from the epistemological to the pedagogical. Some are considerably more controversial than others. Not all of the claims are equally, or at all, accepted by all constructivists.

1. Knowledge is constructed by learners, it is not (and cannot be) transmitted as is.
2. The knowledge that people – or groups of people – have is different from the knowledge of other people who have ostensibly 'learned the same thing'.
3. Knowledge construction takes place as prior knowledge interacts with new experience.
4. Knowledge is not derived from, is not about, and does not represent an external, observer-independent natural reality, but an experiential, personal (or socially shared) 'reality'.
5. If an extraexperiential reality exists, it is not rationally accessible.
6. There is no objectively correct or incorrect, true or untrue knowledge, only knowledge that is more or less viable for a purpose.
7. The social and cultural context mediates the construction of knowledge.
8. Effective learning features the learner as an intellectually active constructor of knowledge; good teaching means facilitating and motivating such construction.
9. Allowing students to leverage their prior knowledge is key to good teaching.
10. Learners should be presented with minimal information and allowed to discover principles and rules for themselves.
11. Hands-on learner-directed exploration is an effective form of learning.
12. Effective learning requires complex, authentic learning situations. Learners should solve ill-structured, open-ended problems similar to those that experts solve.

13. Effective learning has a social dimension, e.g., groupwork.
14. Education should not impose learner-independent norms or goals; the learner is in charge.
15. It is not possible to, or does not make sense to, apply standard evaluations to assess learning.

The following sections elaborate on these claims. Section 6.1 below gives an overview of the main types of constructivism. In Section 6.2, I describe in more detail some of the main features of constructivist epistemology. The discussion in that section revolves largely around the more extreme forms of constructivism; I outline a moderate position in Section 6.3. Section 6.4 provides a brief overview of constructivist pedagogy. Sections 6.5 and 6.6 introduce two relatives of constructivism, conceptual change theory and situated learning, respectively. In Section 6.7, we get to computing education and the impact of constructivism on CER. Finally, Section 6.8 reviews some of the not insignificant criticism that constructivism has drawn.

6.1 There are many constructivisms

The roots of constructivism go well back in time. John Locke was an early influence, and Immanuel Kant is sometimes mentioned as one of the first real constructivists. Many of John Dewey's progressivist ideas are echoed in present-day constructivism. Especially through the seminal work of Jean Piaget and Lev Vygotsky, constructivism has become a major force in education since the 1900s. Subsequently, an ever increasing number of different interpretations of constructivism have come into existence. Gunstone (2000, p. 254) even suggests on the basis of an internet search that "there are no areas of human activity to which the label "constructivist" is not currently being applied in some form!". Phillips (2000, p. 7), who sets out to describe the "constructivist landscape", characterizes the view as "nightmarish" in reference to the daunting task of making sense of the myriad interrelated, sometimes complementary, sometimes contradictory constructivisms in the literature.

6.1.1 Nowadays, everyone is a constructivist(?)

It is nigh on impossible to find a present-day educational researcher that believes that learning simply involves the transmission, or 'pouring', of pre-existing knowledge from a teacher or a book into students. Calling oneself a constructivist is politically correct; denying the active role of learners in building knowledge is to invite scorn. "There is a very broad and loose sense in which all of us these days are constructivist" (Phillips, 1995, p. 7). However, we vary in how constructivist we are, and in how we are constructivists.

How constructivist are you?

Phillips (1995, 2000) characterizes constructivists by considering their positions along a dimension that is essentially a measure of how much one emphasizes the idea of knowledge as a construction. That is, to what extent does one consider knowledge to be a reality-independent, perspectival human construction as opposed to something that is dictated by nature itself, i.e., by what is real? In this scheme, those who espouse extreme versions of constructivism are at one end, and non-constructivists at the other, with the middle of the constructivist landscape occupied by "a marsh of wishy-washy scholars". Wishy-washy constructivists prefer watered-down, non-radical versions of constructivist epistemology.[1] Like Phillips, I, too, find myself in these wetlands. (Section 6.8 discusses the reasons.)

A popular buzzword

Constructivism has become a common buzzword that features in numerous publications whose basis in constructivist learning theory or epistemology is debatable. A colleague of mine recently made what I think was a more-than-half-serious comment to the effect that "Nowadays, you may need to put some

[1] This should not be interpreted as being wishy-washy about the importance of epistemological questions.

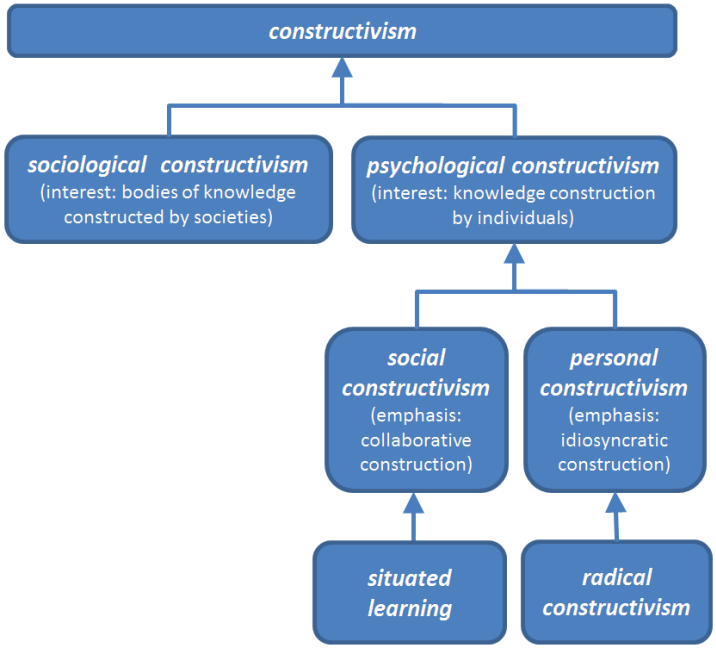


Figure 6.1: A classification of constructivisms, roughly based on Phillips (1995, 2000).

learning theory into CER papers. But don't worry – just mention constructivism and the reviewers will be happy."[2]

Matthews (2000) observes that some educationalists apply the term "constructivism" to any non-behaviorist learning theory, or to any view that recognizes social, cultural, and historical aspects of cognition. Many practitioners are not aware of the differences between forms of constructivism, or indeed about the underlying epistemological and ontological assumptions of whichever form of constructivism they profess allegiance to (Prior McCarty and Schwandt, 2000). These practitioners are not decidedly wishy-washy; they are uncommitted and may retain a traditional epistemology merely by default.

These developments make the term increasingly ambiguous and hinder the debate on the deeper issues involved. This has understandably made many authors unhappy, as they would prefer to reserve the term for epistemologically explicit variants of constructivism. This group includes critics of constructivism, wishy-washy constructivists, and proponents of extreme positions such as Ernst von Glasersfeld, who coined the term *radical constructivism* specifically to differentiate his views from "naïve constructivism" that does not question traditional epistemology.

6.1.2 Constructivism comes in a few main flavors

Buzzwords aside, there are dozens of variants of epistemologically grounded constructivism (see, e.g., Matthews, 2000; Ernest, 1995, and references therein) that I will not even attempt to cover. However, constructivisms can be roughly grouped by their main emphasis (Phillips, 1995, 2000). Figure 6.1 is a breakdown of some of the main branches of constructivism.

A constructivism is typically concerned with one (or sometimes both) of two things: 1) sociological issues: the construction of bodies of knowledge by societies at large (e.g., bodies of scientific knowledge), or 2) psychological issues: the construction of individuals' knowledge. Constructivist views on sociology have been seminally influenced by Kuhn (1962) and taken to an extreme by the so-called Edinburgh strong

[2]It may not be as simple as that. This is amusingly illustrated by Derry's (1996, p. 172) anecdote, in which constructivist journal editors pick on an eminent Immanuel Kant Professor of Psychology for his use of words such as "say", which supposedly suggest a knowledge-transmission view of learning, despite the fact that the professor and his academic ancestors "were constructivist when constructivism was not cool".

programme led by David Bloor (see, e.g., Phillips, 2000). They are of primary interest to sociologists and philosophers of science but less pertinent for my present purposes. I will largely focus on *psychological constructivism*, which applies more directly to education.

Psychological constructivisms can be divided into *personal constructivisms* and *social constructivisms*[3]. Personal constructivists emphasize the idiosyncratic construction of individual knowledge. Social constructivists instead emphasize the importance of the social and cultural nature of individuals' knowledge construction. Radical constructivism, the influential extremist version of personal constructivism fathered and fronted by von Glasersfeld (e.g., 1982, 1995, 1998), is worthy of separate mention as it is a staple of constructivist literature that has been claimed to represent "the state of the art in epistemological theories for mathematics and science education" (Ernest, 1995, p. 475). *Situated learning*, which also appears in Figure 6.1, is a specific form of social constructivism; I will say more about it in Section 6.6.

6.2 Knowledge is an (inter-)subjective construction

The subsections below briefly outline constructivist positions on learning, epistemology, and educational goals.

6.2.1 We learn by combining prior knowledge with new experience

Two ideas are central to most constructivist views of learning (Howe and Berv, 2000).

1. The learning of something new builds on the learner's existing knowledge and interests that learners bring into the learning context.
2. Learning is the construction of new understandings through the interaction of the existing knowledge and new experience.

These ideas are very generic. Precisely what kinds of processes take place as prior knowledge interacts with new experience, and what kinds of conceptual structures are constructed as a result, are not something that constructivists agree on. Some constructivists do not even look for a more specific description, sometimes because they hold the view that not only is everyone's knowledge different, but also the way in which construction takes place and the nature of the resulting conceptual structures are idiosyncratic and cannot be universally described. So-called cognitive constructivists seek to reconcile constructivist ideas with the information-processing views of cognitive science (e.g., Derry, 1996) and use, e.g., schema theory and mental model theory to explain construction. Conceptual change theories (Section 6.5 below) are also espoused by some constructivists.

While the above principles of learning are readily accepted by many who do not call themselves constructivists, constructivist views on epistemology are not all equally uncontroversial.

6.2.2 Do we all have our own truths?

The traditional, commonsense view of ontology is that there exists a real world that is independent of our understandings of it. Traditional, commonsense epistemology suggests that knowledge reflects an existing reality, and the truth value of a proposition depends on its correspondence with said reality.

Constructivism questions the very notions of objective knowledge and truth. For the full-blooded constructivist, an objective reality is unknowable, if there is one at all. Von Glasersfeld makes a distinction between ontological reality and each individual's lived, experiential reality. Only the latter is relevant:

> *for the constructivist, 'existence' must not be interpreted ontologically but epistemologically. That is to say, it refers to the realm of cognitive operating and structuring, and not to the realm of 'being' in the traditional sense.* (von Glasersfeld, 1982)

[3]The label "social constructivism" is variously attached to one or both of what I call "social constructivism" and what I call "sociological constructivism" (and some authors do write simultaneously on both sociological and psychological issues). In addition, the term "social constructionism" is used in a number of ways. I will steer clear of the latter term and use "social constructivism" to mean those constructivisms that have an interest in psychology and emphasize social aspects of individuals' knowledge construction.

Social constructivist reasoning takes a different path, but the conclusion is the same; for instance, Gergen describes his (extreme) social constructivism as ontologically "mute" (Prior McCarty and Schwandt, 2000).

No objective knowledge

According to radical constructivists, knowledge is non-foundational and non-representational. It is not a copy, a reflection, or a representation of an external, observer-independent ontological reality. Instead, all that we call knowledge is derived – constructed – from the lived, experiential world of the individual mind (e.g., von Glasersfeld, 1998). Social constructivists disagree with radical constructivists' emphasis on the individual, but share with them a key epistemological claim. According to extreme proponents of both branches of constructivism, knowledge refers only to the individual's (personal constructivism) or the social group's (social constructivism) constructions.

For the radical constructivist, knowledge is idiosyncratic and "no grounds exist for believing the conceptual structures that constitute meanings or knowledge are held in common among different individuals [...] and one can never say whether or not two people have produced the same construct" (Howe and Berv, 2000). For the social constructivist, on the other hand, knowledge *is* shared; sharing is in fact what creates knowledge, which is defined as "temporal locations in dialogic space – samples of discourse that are accorded status as "knowledgeable tellings" on given occasions" (Gergen, 1995, p. 30). According to this interpretation, whether something counts as knowledge does not depend on an external reality but on whether the participants in a 'language game' jointly grant that something the status of knowledge. Any knowledge so constructed by those participants is particular to that game and incommensurable with others' knowledge.

No truth, only viability

Since individuals' or groups' knowledge can be compared neither to that of other individuals or groups, nor to an ontological reality, constructivism in its more extreme forms abandons the notions of correct and incorrect, true and false. All knowledge is individually or socially constructed, imperfect, and fallible.

Many constructivists think of knowledge less as something that is possessed than as the ability to accomplish something in a particular context. Von Glasersfeld (1995) exhorts: "Give up the requirement that knowledge represents an independent world, and admit instead that knowledge represents something that is far more important to us, namely what we can *do* in our *experiential world*" (pp. 6–7, original emphasis). At the social constructivist extreme, being knowledgeable is conceived as a status given by participants in a language game to one of their number: "For the purposes of the conversation, 'I know' when I speak in ways that enable you to treat me as if I know, and vice versa. We successfully generate dialogue because we are mutually accorded the status of knowledgeables across time." (Gergen, quoted by Prior McCarty and Schwandt, 2000, p. 58)

This procedural view of knowledge affords the notion of *viability*, the radical constructivist's stand-in for truth. Viability refers to the pragmatic adequacy and task-relevance of knowledge, as determined by the knower.

> *Simply put, the notion of viability means that an action, operation, conceptual structure, or even a theory, is considered "viable" as long as it is useful in accomplishing a task or in achieving a goal that one has set for oneself. Thus, instead of claiming that knowledge is capable of representing a world outside of our experience, we would say* [...] *that knowledge is a tool within the realm of experience.* (von Glasersfeld, 1998)

For instance, knowledge of a programming language that allows a particular bug-free program to be written is viable for that task. The same knowledge may not be viable for the task of writing a different program. Different programmers have possibly vastly different conceptual structures that may or may not match the defined standard of the programming language and its technical implementation. However, for the radical constructivist, such judgments are beside the point as long as one's knowledge is viable for one's intended purposes. Only when one's knowledge is inadequate is it necessary to construct a new understanding.

Context-dependence of knowledge

The constructivist claim goes beyond stating that the viability of knowledge – that is, the usefulness of the conceptual structures we have constructed – is context-bound. In the constructivist view, the content and the very nature of our conceptual structures depend on the contexts in which we created them. Knowledge, then, is not only specific to each individual or group, but to learning contexts as well.

Many constructivists emphasize the difficulty of transferring knowledge from one context to another. Achieving viability in one context is barely any guarantee of viability in another kind of context. An example is given by Greening (1999, p. 50), who discusses the knowledge students construct in traditional lectures. According to Greening, such knowledge may only be useful (viable) in the "fairly artificial and constrained" educational system itself – in an exam, for instance – rather than in real-life situations that call for it.

These views on epistemology have led constructivists to oppose traditional education.

6.2.3 Is it impossible to standardize any educational goals?

In traditional views of education, teachers – as possessors of more true knowledge than learners – are in a position to decide what kind of knowledge learners should acquire. The job of learners is to receive true knowledge from their teachers and from books. Whether they succeed can be summatively measured. Constructivists – personal and social alike – challenge this view, questioning the primacy of the teacher, the setting of educational goals that are the same for each student, and the meaningfulness of standardized assessment.

From the radical constructivist perspective, the knowledge of a teacher or a curriculum-writer speaks merely of their own experiential reality; social constructivists like Gergen likewise reject claims of teachers' authority and consider teachers and learners to be equal participants in the social constitution of knowledge (see, e.g., Gergen, 1995; Prior McCarty and Schwandt, 2000). Educators are not in a privileged position that would justify externally setting educational goals for others; instead, the goals of education must emerge from the learners themselves, their interests, and their interactions with new experiences and with other people. To the fully committed constructivist, knowledge is not content that exists independently of the learner; correspondingly, constructivist education is less about covering content than it is about developing a range of views on interesting matters, with everyone involved serving as both learner and teacher.

Reassessing assessment

The way educational goals elude standardization naturally carries over to the constructivist view of assessment. Many constructivists maintain that since knowledge and viability are context-dependent, assessment should be based on fitness for purpose rather than correspondence with 'fact'. Furthermore, since the goals of education are largely learner-set, and viability learner-determined, it is the learners themselves who are primarily responsible for judging whether they have reached their goals (Greening, 1999). In the social perspective, viability is assessed via social negotiation that compares one's knowledge with that of others, with the learning community serving as a test bench.

This view of assessment is in sharp contrast with the traditional objectivist view. It is clearly not accepted by all constructivists. Larochelle and Bednarz (1998), who adopt a strongly constructivist stance, are not satisfied with those self-proclaimed constructivist educators who nominally accept the fact that learners construct their own knowledge, but fail to take constructivist thinking to its conclusion; it is only accepted that many roads lead to Rome, but they all still lead to Rome. To extend Larochelle and Bednarz's metaphor, the radical constructivist view is that the Eternal City might not even exist; rather, learners construct their own experiential Romes. If the learner is satisfied that they have got a satisfactory peek at the Coliseum, then they have reached a viable destination (for now).

Figure 6.2: A radical destructivist (de Lipman, 1897).

6.3 But a wishy-washy constructivist also acknowledges reality

'Wishy-washy' constructivists generally accept some form of the realist proposition that a world exists, and search for a reasonable epistemological middle ground that avoids completely jettisoning the concepts of truth and reality while at the same time allowing for subjectivity. According to Phillips (1995, p. 12), "any defensible epistemology must recognize [...] the fact that nature exerts considerable constraint over our knowledge-constructing activities, and allows us to detect (and eject) our errors about it". One possible basis for such an epistemology was provided by Popper (1972), who suggested an ontology of three interacting 'worlds'. $World_1$ is a world of physical objects. $World_2$ consists of psychological, subjective experiences constructed by people of the other two worlds. $World_3$, which is dependent on the first two worlds, contains the products of the human mind, such as art, mathematics, values, and science. The latter exists as a result of human activity and is maintained by it, rather than existing as an ideal world that transcends $World_1$ and $World_2$ as Plato would have it. Popper's is essentially a moderate constructivist view (as discussed, e.g., by Harlow et al., 2006; Niiniluoto, 1980).

Regarding goals and assessment, a wishy-washy constructivist might say that while everyone has their own Rome and their own Appian Way, those constructions and their usefulness are affected by the location of the actual city. Ideally, experiential Romes should be reasonably close to the real one, with all roads leading at least to its near vicinity.

6.4 For better learning, constructivisms tend to encourage collaboration in authentic contexts

> *Where the traditional teacher-centered direct instruction is the norm in teaching, there exists a paradox:* "The more you teach, the less they learn." [...] *the time the teacher spends on direct instruction takes away from the dialogue, negotiation, debate and assessment that could take place between students.* [...] *Learning in its constructivist sense therefore requires us*

> *to reduce teaching that is based on direct instruction and to emphasize the social interaction between learners and personal reflection.* (Sahlberg, 1996, p. 79, my translation)

Constructivist pedagogy encourages teachers to engage the learner in order to to help them construct their own knowledge and to take individual differences and prior knowledge into account. Constructivism is generally associated with a move away from traditional pedagogies in which the learners' role is relatively passive. Some constructivists vehemently oppose lectures in particular. Decontextualized teacher-given practice tasks devoted to narrow topics (such as those found in many textbooks) are also criticized as 'drill and kill'. For many constructivist teachers, engagement implies hands-on tasks.

Even bearing in mind that there are different varieties of constructivism, a general trend can be observed in that constructivists tend towards learning environments that are situated in (or emulate) complex, realistic contexts, and that require students to solve problems. Social constructivists in particular (but personal constructivists as well) emphasize social interaction in learning environments: students collaborate, negotiate, and compare their ideas with those of the other students and the teacher. Some constructivists advocate using ill-structured problems that students need to make sense of on their own or in groups, with limited guidance from teachers. This form of constructivist pedagogy can be considered as a form of *discovery learning* (e.g., Bruner, 1979; Papert and Harel, 1991), as it leaves students to discover solutions on their own – this is a hotly debated issue that I will return to in Section 6.8 below.

Problem-based learning (PBL) is a currently popular pedagogy that exemplifies many constructivist principles (see, e.g., Savery and Duffy, 1995; Norman and Schmidt, 1992). PBL is driven by substantial, realistic problems that groups of learners try to solve with limited help (and no ready-made solutions) from teachers and tutors:

> *The principal idea behind problem-based learning is that the starting point for learning should be a problem, a query or a puzzle that the learner wishes to solve... Problem-based courses use stimulus material to engage students in considering a problem which, as far as possible, is presented in the same context as they would find it in 'real life;' this often means that it crosses traditional disciplinary boundaries. Information on how to tackle the problem is not given, although resources are available to assist the students to clarify what the 'problem' consists of and how they might deal with it. Students work cooperatively in a group or team with access to a tutor who is often not an expert in the field of the particular problem presented, but someone who can facilitate the learning process.* (Boud and Feletti, quoted by Kay et al., 2000)

PBL seeks to activate students' prior knowledge in the search for solutions to meaningful problems, to encourage the sharing of cognitions amongst learners and (self-)explanation of solutions, and to foster the development of self-directed learning skills (Norman and Schmidt, 1992).

Inquiry-based learning (IL) is another popular constructivist approach to teaching that is closely related to PBL – indeed, it is often almost indistinguishable from it (Hmelo-Silver et al., 2007). Whereas PBL arises from the diagnosis problems of medical education, IL draws on the scientific method: students form questions, then collect and analyze data to answer them.

Despite the emphasis on hands-on activity and methods such as PBL and IL, many constructivists are careful to point out that using a particular method has no intrinsic value – a lecture can be constructivist if it succeeds in engaging learners to construct viable knowledge, just as richly contextualized learner-driven groupwork fails when inappropriately used.

6.5 Conceptual change theories deal with the dynamics of knowledge construction

This section briefly introduces a group of learning theories related to constructivism. Depending on interpretation, these *conceptual change theories* can be said to be either versions of constructivism or separate theoretical frameworks that have considerable commonalities with forms of constructivism. They

certainly fall under the broad usage of the term "constructivism" (see Section 6.1).[4]

Theories of conceptual change characterize the kinds of conceptual structures that people have, and when and how those structures change as we learn. Of particular interest to conceptual change theorists are people's intuitive, naïve (as opposed to scientific) conceptions of phenomena. Problematically for the educator, as well as the learner, the naïve knowledge that arises from everyday experience is notoriously resilient to change.

Conceptual change theories generally fall into one of two 'families', termed by Özdemir and Clark the *knowledge-as-theory* and *knowledge-as-elements* perspectives (Özdemir and Clark, 2007; diSessa, 2006).[5]

Knowledge-as-theory

One well-established family was seminally influenced by the work of Posner, Strike, and McCloskey (and their colleagues) which in turn draws on Kuhn's paradigm shifts and the Piagetian notion of accommodation (see, e.g., Posner et al., 1982; Strike and Posner, 1985).

Knowledge-as-theory perspectives liken individuals' knowledge – even naïve knowledge – to scientific theories. What this means is not that individuals are necessarily aware of their knowledge in the same way as scientists are of their theories, or that they seek to verify it as scientists do. The similarity to scientific theory is merely that even naïve knowledge is considered to be organized and coherent, to allow consistent application across contexts, and to be replaceable by a new 'theory' when it is found lacking.

When a learner experiences something new, they relate it to their existing ideas and judge its consistency with them. As existing concepts form coherent structures, they are not independent of each other; changes to one concept require changes in others. The difficulty of getting rid of misconceptions is explained by this interdependency: making broad, complex changes to one's existing overall 'theory' – misconceived or otherwise – is a daunting task. However, when sufficient conditions are met, a revolutionary change akin to a scientific paradigm shift (Kuhn, 1962) occurs in the conceptual ecology as one's old 'theory' is replaced by a new one. A revolution is not necessarily abrupt, but may occur gradually (Posner et al., 1982). Such a revolution does not result in anarchy; on the contrary, it restores the conceptual ecology to a state where one's new 'theory' accommodates the disruptive new ideas that previously failed to fit in.

From a practical point of view, the meat of knowledge-as-theory perspectives is in what they say about the conditions under which conceptual revolutions occur. Posner et al. (1982) detail four primary conditions for conceptual change: 1) dissatisfaction with existing conceptions; 2) a sufficient minimal understanding of the new conception; 3) the apparent plausibility of the new conception (it must seem on the surface as if it could solve problems that one's existing conception does not), and 4) an apparent possibility that the new conception will be fruitful in leading to new insights beyond present needs. Many learning events start with anomalies that lead to dissatisfaction and increase the plausibility and apparent usefulness of new conceptions. Anomalies "provide the sort of *cognitive conflict* (like a Kuhnian state of "crisis") that prepares the student's conceptual ecology for an accommodation" (Posner et al., 1982, p. 224, my italics). From a knowledge-as-theory perspective, teachers should foster cognitive conflict that leads learners to confront and supplant their misconceptions.

Knowledge-as-elements

Another family of conceptual change theories, championed by Andrea diSessa, lies on the other side of the "fault line between coherence and fragmentation" (diSessa, 2006). According to these theories, naïve knowledge is not coherent and consistent but consists of a collection of quasi-independent, often tightly context-bound elements that form a loosely connected structure. In diSessa's influential knowledge-as-elements theory, these low-level elements of naïve knowledge are called *p-prims* (phenomenological primitives). A p-prim is a minimal abstraction of common phenomena that is – for whoever experiences

[4] I have placed this discussion of conceptual change theories in this chapter on constructivism. This body of work also intersects significantly with cognitive psychology, as indeed does constructivism in general (assuming a broad definition of the latter term).

[5] In the literature, it is not rare for the term "conceptual change theories" to be used to refer exclusively to what I call knowledge-as-theory perspectives.

it – intuitive and self-evident. An example of a p-prim in naïve physics is the notion of 'bouncing': as a smaller object comes into impingement with a large or otherwise immobile other object, the smaller object will recoil (diSessa, 1993). 'Bouncing' is a common-sense equivalent of a physical law: it helps explain other phenomena but does not itself, as a primitive, require explanation.[6]

Knowledge-as-elements perspectives emphasize context-dependence. There is no overarching theory-like structure that allows for generalization. As the learner forms a new idea, it becomes an element in the learner's conceptual ecology, associated with a particular context. It is possible for a learner to hold several contradictory understandings of a phenomenon, each particular to a different context. Instead of having a single consistent theory for the many physical phenomena governed by $F = ma$, for instance, one may have entirely distinct p-prims for bouncing and throwing, and even for throwing objects of a certain shape.

Marton (1993) provides a nice summary of diSessa's view of learning:

> *This view of characterizing what it takes to learn physics contradicts other characterizations in which the transition between naive and scientific physics has been seen in Kuhnian terms as the replacement of one world view with another.* [...] *The picture presented by diSessa is more evolutionary than revolutionary: Scientific physics evolves from naive physics more by organization than by reorganization, more by structuring than restructuring.* (p. 228)

According to knowledge-as-elements perspectives, it is context-specificity that makes misconceptions resilient to change; instruction intended to correct a misconception may simply add a parallel understanding rather than replace the old. However, some proponents of knowledge-as-elements maintain (from an explicitly constructivist position) that rather than being a problem, misconceptions are a useful or even necessary basis for further learning (e.g., Smith et al., 1994). Although Smith et al. agree with knowledge-as-theory perspectives that states of cognitive conflict are "certainly desirable and conducive to conceptual change" (p. 22), they oppose the idea of *confronting* misconceptions and trying to replace them with correct understandings. The challenge of learning from such a knowledge-as-elements perspective is not to replace the learner's naïve misconception-ridden understanding with another – there is no well-organized, general structure to replace! – or even to replace individual misconceptions. Instead, the goal is to help the learner to organize their knowledge elements, to find borders for the rules they have constructed so as not to over- or under-generalize, to select the most productive ideas and refine them, and to observe similarities across contexts. The elements of naïve knowledge – misconceptions included – are the raw materials for such processes, which, if learning is successful, evolve into a theory-like scientific understanding.

Recent developments have seen knowledge-as-theory and knowledge-as-elements move somewhat closer to each other, with researchers suggesting that both evolutionary and revolutionary changes are important during learning and that both families of conceptual change theory can inspire improvements in pedagogy (see, e.g., Özdemir and Clark, 2007; Hammer, 1996).

6.6 Situated learning theory sees learning as communal participation

This section introduces another theory that is a relative or form of constructivism, depending on interpretation. The theory of *situated learning* conceives of learning as an intrinsically contextualized process of social participation in a community (Lave and Wenger, 1991). Although it is not always phrased in such terms, situated learning theory is based on a social constructivist epistemology in which knowledge exists within a community rather than within individuals.

Ben-Ari (2004, pp. 86–87) primes us on the lexicon of situated learning, as defined by Lave and Wenger:

> *The learner is considered to be a participant within a* community of practice *(CoP). Learning occurs by a process of apprenticeship called* legitimate peripheral participation *(LPP): (a) the learner* participates *in a community of practice, (b) the learner's presence is* legitimate

[6] Although they were concerned with societal phenomena rather than physical ones, and their approach was less theoretical, Run-D.M.C. (1984) might as well have been rapping about p-prims as they cautioned would-be inquirers: "Don't ask me, because I don't know why, but it's like that, and that's the way it is."

in the eyes of the members of the community, and (c) initially, the learner's participation is peripheral*, gradually expanding in scope until the learner achieves full-fledged membership in the community.*

Situated knowledge

> *The place of knowledge is within a community of practice.* (Lave and Wenger, 1991, p. 100)

Situated learning theorists, like constructivists in general, emphasize the contextualization of knowledge and difficulties in transfer. Knowledge is seen as located – situated – in the lived world of concrete practice in which it is put to use, and tied to that cultural, social, and physical context. All knowledge is situated in some way or another, as "a community of practice is an intrinsic condition for the existence of knowledge" (ibid., p. 98). Conversely, the importance of individuals, even the recognized 'masters' or teachers within a community, as bearers of knowledge is of lesser importance: "mastery resides not in the master but in the organization of the community of practice of which the master is part" (ibid., p. 94). This is a quintessentially social-constructivist view.

Learning is participation

According to Lave and Wenger, learning occurs as the learner participates in the practice of a community in some useful – although initially minor, even menial – capacity.

> *Participation in the cultural practice in which any knowledge exists is an epistemological principle of learning. The social structure of this practice, its power relations, and its conditions for legitimacy define possibilities for learning (i.e., for legitimate peripheral participation).* (Lave and Wenger, 1991, p. 98)

Participation does not mean mere observation and imitation of what the senior members of the community do, but taking an active, social, useful, collaborative role within the community that enables the learner to take part in the community's everyday activities, becoming familiar with all its aspects, including "who is involved; what they do; what other learners are doing; [...] how masters talk, walk, work, and generally conduct their lives; [...] how, when, and about what old-timers collaborate, collude, and collide, and what they enjoy, dislike, respect, and admire" (ibid., p. 95). The other members of a community are exemplars of social roles and professions that learners can aspire to. The learner's gaining of expertise is embodied in changes that take place within the community: the learner gradually progresses from the periphery to ever more central roles, eventually reaching mastery.

The problem of decontextualized schooling

Since learning and its goals are characteristics of a community of practice, Lave and Wenger (1991, p. 97) advise against analyzing curricula "apart from the social relations that shape legitimate peripheral participation". According to them, engaging in the practice of a community is not merely the future goal of a learning process (as it is traditionally seen in the context of schooling that prepares students for later participation in other, professional communities) but is an intrinsic aspect of learning. In their seminal work, Lave and Wenger (1991) deliberately skirt around questions of pedagogy, but, as a whole, the situated learning movement is severely critical of the notion of teaching knowledge that is abstracted from authentic practice, as "the organization of schooling as an educational form is predicated on claims that knowledge can be decontextualized" (ibid., p. 40).

In the situated view, all learning is situated, and useful results are only likely to be obtained by situating learning in the community of practice in which the skills that are learned are genuinely needed. Participating in a traditional school community, for instance, will only (or mostly) foster skills that are needed in that inauthentic setting; "there are vast differences between the ways high-school physics students participate in and give meaning to their activity and the way professional physicists do" (ibid., p. 99) – for the purpose of becoming physicists, they are participating in the wrong community of practice.

The challenge of the educational enterprise, from the situated learning point of view, is to find ways for learners to become members of meaningful communities of practice and to engage in legitimate, useful participation right from the start. The key to this is access for newcomers to all that membership of the community entails: ongoing activity, old-timers and other members, information, resources, and opportunities for participation. Lave and Wenger point out that although this is essential, it is always problematic all the same.

6.7 Constructivisms are increasingly influential in computing education

> *We feel that constructivist principles are exerting strong influences on professional practice in computer science education. However, constructivism – as a body of theory – maintains a relatively low visibility within our discipline.* [...] [Constructivism] *has spawned a host of principles for good practice that have propagated independently of theoretical roots.* (Greening and Kay, 2001)

According to Greening (1999), "a constructivist future for computer science education is almost assured". Be that as it may, the influence of constructivisms has indeed become more explicit in both computing education and CER since the late 1990s.

Compared to its impact on mathematics and science education, the visible influence of constructivisms in computing education has been minor. Certainly, there are famous initiatives in programming education that are based on a kinstheory of constructivism that goes by the name of *constructionism* (Papert and Harel, 1991). The 'toolkits-for-learning-by-creating-for-sharing' spirit of constructionist education is embodied in the Logo and Smalltalk programming languages, and, more recently, Scratch (MIT Media Lab, n.d.), Squeak (Ingalls et al., 1997), Lego Mindstorms (Lego Group, n.d.), and Media Computation (Media Computation teachers, n.d.). Equally certainly, many computing education practices are compatible with many forms of constructivism. For instance, project-based teamwork on open-ended problems that are intended to be fairly authentic is relatively common in programming education.

However, it is only in recent years that constructivism has become a clearly visible force in CER. The use of "constructivism" (as a buzzword, or as a genuine theoretical framework) is increasingly common and constructivism in some form is now used to justify and inform pedagogical interventions in programming education (Hadjerrouit, 1998; Gray et al., 1998; Van Gorp and Grissom, 2001; Parker and Becker, 2003; Lui et al., 2004; Gonzalez, 2004; Wulf, 2005; Thota and Whitfield, 2010; Yadin, 2011, and many more), to motivate research questions or approaches (e.g., Madison and Gifford, 1997; Rajlich, 2002; Knobelsdorf, 2008; Ma et al., 2011), or as an analytical tool that retroactively justifies existing pedagogy (Pullen, 2001). Some recent work in CER has explored pedagogy that builds on the notion of legitimate peripheral participation (e.g., Hundhausen, 2002; Guzdial and Elliott Tew, 2006) or otherwise used situated learning as a research framework (e.g., Booth, 2001b; Knobelsdorf and Schulte, 2007). The related constructivist approach of cognitive apprenticeship, which seeks to make thinking processes visible (Collins et al., 1989) has also found some recent applications in computing education (e.g., Caspersen and Bennedsen, 2007; Bareiss and Radley, 2010), as has problem-based learning (e.g., Kay et al., 2000; Kinnunen and Malmi, 2005; Nuutila et al., 2005, and see Section 10.1).

The vast majority of CER papers that mention "constructivism" are concerned directly with pedagogy. Only a few authors have considered the applicability of a constructivism as a theory to computing education from a wide perspective. The discussion in two such publications, by Greening (1999) and Ben-Ari (2001a), forms the spine of Subsections 6.7.1 and 6.7.2 below. In Subsection 6.7.3, I review Ben-Ari's commentary on the applicability of situated learning theory to computing education.

6.7.1 Programmers' jobs feature ill-structured problems in an ill-structured world

Greening (1999) argues that, from the constructivist point of view, the main challenge of learning programming is not the acquisition of knowledge about programming languages, syntax, and semantics. Rather, learners must come to see programming as an essentially creative pursuit involving the skills of synthesizing and problem solving. Genuine skill in programming involves being able to tackle ill-structured, complex problems in authentic contexts. To cope with such contexts, students need to learn the skills

of proper design, coding style, requirements elicitation, software development processes, teamwork, and communication. Greening advocates PBL as a pedagogical technique suitable for learning these skills.

Greening further claims that constructivism is much needed in computer science education to address the modern trends of information explosion, rapid technological change, globalization, and the resulting need to recognize the validity of multiple viewpoints on knowledge. "Education will not be able to assume that a singular world view will provide an adequate working model; it will need to deal instead with multiple world views in flux" (p. 67). This is something that constructivism, with its subjective view of knowledge, is more comfortable with than the traditional objectivist paradigm is.

When it comes to various human aspects of computing, such as software engineering practices, good design, usability, etc., Greening is no doubt right to stress the usefulness of learning to cope with multiple viewpoints. However, when it comes to learning how to cope with the computer itself, a different line of constructivism-inspired thinking is suggested by Ben-Ari (2001a). The computer does not negotiate...

6.7.2 The novice needs to construct viable knowledge of the computer

> *Even if no effort is made to present a view of what is going on 'inside' the learners will form their own.* (du Boulay, 1986)

Ben-Ari (2001a) discusses the application of constructivism – primarily personal constructivism of a cognitivist bent – to computing education, drawing a number of conclusions that "seem to follow directly from constructivist principles". Ben-Ari's conclusions rest on three claims. The first is that learners necessarily construct knowledge about the phenomena they encounter, for better or worse. The other two are characteristics of computing as a discipline:

> *I claim that the application of constructivism to CSE must take into account two characteristics that do not appear in natural sciences:*
>
> - *A (beginning) computer science student has no effective model of a computer.*
> - *The computer forms an accessible ontological reality.*
>
> (Ben-Ari, 2001a, p. 56)

Let us look at each of the three claims in more detail, starting with students' initial models.

Lack of an effective initial model

Constructivism emphasizes the importance of prior knowledge for learning, and knowledge of the computer is, Ben-Ari (2001a) argues, a prerequisite for understanding computing as we know it. However, beginning students of computing lack knowledge that "the student can use to make viable constructions of knowledge based upon sensory experiences such as reading, listening to lectures and working with a computer". That is, students lack knowledge that is viable for the purpose of learning about programming and other aspects of computing.

Some students come to computing studies with self-taught, viable knowledge of the computer and of programming concepts. However, prior knowledge may also be a hindrance: "Autodidactic programming experience is not necessarily correlated with success in academic computer science studies. These students, like most physics students, come with firmly held mental models that are not viable for academic studies" (ibid., p. 58). Depending on which variant of constructivism one subscribes to, dealing with such non-viable prior knowledge is either a matter of correcting it or of refining it to create coherent, viable knowledge (cf. Section 6.5).

Inevitable construction

Ben-Ari (2001a; Ben-Ari and Yeshno, 2006) argues that, according to constructivist principles, when learners come across a system they construct a mental model of it. Constructing knowledge is inevitable and will happen regardless of whether the learner has been taught a normative conceptual model of the

phenomenon, although instruction can have an effect on the resulting model. For instance, a person encountering a WYSIWYG word processor for the first time may construct a model based on an analogy with paper and ink, which is viable to begin with but may soon become non-viable once one starts tinkering with fonts (which are implemented as invisible markup within the visible document). A technical description of how a word processor actually works may contribute towards the construction of a different model.

Ben-Ari (2001a) marshals evidence from the literature – much of which is familiar to readers of this thesis from Section 3.4 and Chapter 5 – to support his argument that when faced with abstractions of the computer, students will necessarily construct their own, often non-viable mental models of what lies beneath. Ben-Ari points out that even though these difficulties with are sometimes attributed to features of particular languages, the phenomenon of constructing non-viable models of the computer does not seem to be constrained to any single language or programming paradigm.

Accessible ontological reality

> *By* accessible ontological reality*, I mean that a "correct" answer is easily accessible, and moreover, successful performance requires that a normative model of this reality must be constructed.* (Ben-Ari, 2001a, p. 56)

In essence, what Ben-Ari is saying is that computing (the parts of it that directly involve computers, at least) *does* have an ontology that is reflected in useful knowledge; this goes against the radical constructivist rejection of ontology as an epistemological basis. The computer is an artifact that behaves in a certain way, and unless the learner's understandings of the computer are a close enough match with this reality, there will soon be consequences. Feedback on non-viable understandings is often immediate and "brutal" in the shape of error messages, crashes, and bugs. Moreover, while the specifics of people's constructed understandings of the computer are unique, all viable understandings must match the normative understanding fairly closely, leaving little room for disagreement. As Ben-Ari (ibid., p. 58) points out, "there is not much point negotiating models of the syntax or semantics of a programming language" once the decision to use a particular programming language has been made. So much for the "spectrum of views" (Greening, 1999) that constructivist educators generally hope students will explore!

Implication: underlying models early

If Ben-Ari's claims are accepted, introductory programming students need to learn not just what program code does, but also what goes on beneath, within the computer. Leaving the computer as an abstract 'black box' in teaching means that people will rely on the ineffective, intuitive knowledge that they will nevertheless construct. The students' knowledge of the computer need not be 'complete' but must be viable for the purpose of understanding programs. Ben-Ari goes even further in his pedagogical recommendations, suggesting that students need to confront underlying models (of the computer and software artifacts like word processors) *before* learning about the abstractions above. Encountering abstractions first will lead to the ill-fated construction of intuitive models.

Greening (1999) is skeptical about Ben-Ari's claim[7] that a model of the computer is needed for learning computing, and emphasizes that constructivism is less about prescribing what prerequisite knowledge is needed than it is about acknowledging that prior knowledge plays a part in knowledge construction. Greening further argues that a model of the computer will be less important in the (near?) future:

> *Increasingly, perhaps as a sign of maturity of the discipline(s) of computing, the need to understand the machine will dissipate. This statement will surely horrify some readers.* (Greening, 1999, p. 73)

I think Greening's statement (as I interpret it) has an unhorrifying point, but may at the same time miss the point that Ben-Ari was (I believe) trying to make.

[7] Greening did not have a time machine. I have referred here to Ben-Ari (2001a), a longer version of a 1998 paper that Greening (1999) commented on.

> *In any particular course you will be teaching a specific level of abstraction; you must explicitly present a viable model one level beneath the one you are teaching.* (Ben-Ari, 2001a, p. 68)

Ben-Ari is not suggesting that introductory programming students (who primarily target an understanding of programs at program code level) should understand in detail how computer hardware or the operating system works. What he is saying is that given a targeted level of abstraction, a lower-level understanding – slightly lower but still as high as possible – is needed that is viable for the purpose of explaining phenomena at the targeted level. To put this claim differently, and into context: in order to learn about programming at the code level, students need to learn about a notional machine (see Section 5.4) that concretizes the semantics of code constructs.

Greening appears to be talking about the importance of understanding the actual machine at a fairly low level. I agree with Greening that levels of abstraction in programming appear still to be rising. That may indeed mean that the low-level machine is becoming less and less important for introductory programming education. However, unless computing and the nature of program execution change in a dramatic way that I cannot foresee, a viable understanding of the computer "one level beneath" the targeted level will still be needed. The level of abstraction of the notional machine that is needed may increase in the future, along with the level of abstraction of programming itself, but a notional machine will nevertheless be required by would-be computer programmers.[8]

6.7.3 Situated learning theory is 'partially applicable' to computing education

Ben-Ari (2004, 2005) discusses the applicability of apprenticeship-driven pedagogy based on situated learning (Section 6.6) to education in computing and other high-tech fields. He identifies open-source software development as an area of computing where legitimate peripheral participation is vividly demonstrated. However, his overall conclusion is that legitimate peripheral participation cannot be accepted as a general model of computing education as "it simply ignores the enormous gap between the world of education and the world of high-tech CoPs [communities of practice]" (Ben-Ari, 2004, p. 98). Ben-Ari questions, among other things, the improbable notion that corporations (genuine communities of practice) would hire and educate newcomers lacking skills that *can* be taught in a school context. He also understandably bristles at the suggestion that learners should choose, or be chosen, professions and future communities of practice early in life, as is the case in the low-tech communities studied by Lave and Wenger (1991), and notes that situated learning may help perpetuate social class structures.

> *To consistently uphold a policy of real situations* [...] *A class of rich kids could be asked to compute the most efficient route to sail a yacht from Nice to Barcelona, while a class of poor kids could be asked to compute the salary at which it is advantageous to give up welfare and take a job.* (Ben-Ari, 2005, p. 372)

Despite these strong criticisms, Ben-Ari does not entirely dismiss the potential of situated learning theory for high-tech education. Taken "metaphorically" rather than at face value, he argues, situated learning can serve as a source of inspiration for computing teachers and computing education researchers. In particular: we should take it seriously that learning computing – even at school or university – is a step on the way towards students' initiation into authentic communities of practice within industry and academia. Learning to be a programmer, for instance, requires the learner to learn about communities in need of programmers and the authentic practices in which they engage, and to become motivated to join such a community. Ben-Ari further notes the domain-tied nature of most genuine computing communities of practice, and argues that computing education should introduce one or more non-computing domains to learners, even if this does not take place in a truly authentic setting but a simulated one.

[8] Ben-Ari's claim raises the suspicion of an infinite regress. To learn at one level, you first need to learn at the level below. To learn at that lower level, do you first have to learn at one below that, and so forth? Does this lead to the conclusion that learning to program requires one to start at the level of subatomic particles and work upward in abstraction from there? The regress is avoided through the observation that the understanding at "one level beneath" does not need to be as comprehensive as the understanding at the target level. At each successively lower level, the understanding needed may be increasingly vague, serving only the purpose of being viable (that is, 'good enough') for learning about the next level up. At a sufficiently low level, which is perhaps not very low, even a trivially teachable or intuitively formed vague understanding is enough.

6.8 Constructivisms have drawn some heavy criticism

The constructivist landscape is not a tranquil place, but a gory battlefield on which some wage civil war while others defend against foreign attacks and lead crusades against non-believers. Much of the critical literature is highly polemical, with constructivists having been denounced variously as inquisitors, Stalinists, drunkards, necromancers, and – worst of all – behaviorists.

I cannot hope to do justice to the complex dialogue in the literature, but will try to present some of the main issues. I will largely ignore the in-fighting between branches of constructivism, and focus on general criticisms of constructivist theories.

I have organized this section around nine contentious points, or 'skirmishes'.

Skirmish 1: relativism

> *The common constructivist move is from uncontroversial, almost self-evident premises stating that knowledge is a human construction, that it is historically and culturally bound, and that it is not absolute, to the conclusion that knowledge claims are either unfounded or relativist.* (Matthews, 1994, p. 143)

Perhaps the most common criticism of constructivist epistemologies (see, e.g., various articles in Phillips, 2000; Steffe and Gale, 1995) is that they easily lead to a denial of reality and to moral relativism: in a world of one's own construction, it is meaningless to care about others. There is widespread agreement that moral relativism is bad, which is why it is a popular beating stick for critics to wield. Constructivist theorists are keen to escape accusations of relativism. On the radical constructivist side, such claims are called "misunderstandings of constructivism" (von Glasersfeld, 1995), and sometimes dodged by labeling one's own version of constructivism non-dualistic, that is, by asserting that the world and humans' constructions of it are inseparable (see, e.g., Howe and Berv, 2000, and cf. the next skirmish). In a rather similar vein, Gergen claims that extreme social constructivism transcends such issues since it does not commence with the external world as its fundamental concern nor with the internal mental world (which would lead to solipsism) but with language (Gergen, 1995); this defense, however, simply leads to solipsism of a different, social kind, according to its critics (Bickhard, 1995).

It has been argued that escaping the trap of utter relativism requires a notion of shared meanings between individuals. Towards this end, von Glasersfeld claims that even though for the radical constructivist, meanings are not genuinely shared, they may be "taken as shared" by individuals, as beliefs that have more or less viability. Howe and Berv are critical:

> *Is the meaning of "taken as shared" merely "taken as shared"?* [...] *How can radical constructivists talk to each other? Worse, how can they talk to themselves? Are von Glasersfeld's meanings on Monday the same as his meanings on Friday, or does he merely take them to be the same? How could he know?* (Howe and Berv, 2000, p. 33)

Social constructivists are keen on shared meanings but also fail to escape accusations of relativism. Taken to the extreme, social constructivism argues that the very notion of knowledge is merely a matter of social agreement, an epistemologically relativist view which readily leads to ontological relativism as well (see, e.g., Prior McCarty and Schwandt, 2000).

Many words have been said in the relativism debate, but a resolution does not seem near.

Skirmish 2: logical flaws: foundationalism, dualism, and the element of surprise

Perceived flaws in constructivist claims about epistemology include the failure to explain the notions of construction and viability, as well as constructivists' claims of non-dualism.

Extreme constructivists consider themselves antifoundational, with no reality to found knowledge on. Prior McCarty and Schwandt (2000) contest this view on several accounts, contending that "it is patently absurd, even hilarious, to consider describing either von Glasersfeld's views or those of Gergen as "antifoundational"". They argue, for instance, that the very idea of knowledge construction depends on an ontology since for construction to happen there need to be some people and materials whose existence

precedes construction. Therefore, constructivism is actually a foundationalist theory in which every idea is founded on the metaphysics of construction (pp. 71–72).

Prior McCarty and Schwandt (2000, p. 70) point out another problem with constructivism: its failure to explain surprise. For instance, it is possible for a seemingly healthy person to be surprised when a doctor tells them that they have only two months left, which seems strange if our experiences are independent of ontological reality. According to Prior McCarty and Schwandt, denial of surprise follows from the ontologically agnostic positions of the radical forms of personal and social constructivism. An extension of this argument is that since it is possible for our understandings unexpectedly to turn out to be non-viable, then there must be an ontological reality 'out there' that leads to such developments. The notion of viability therefore appears to lead to an acceptance of ontology (see also Marton and Booth, 1997, pp. 6–8). Arguments of this kind have prompted the suggestion that radical constructivism is a dualist theory, unlike its proponents claim. Dualism comes with baggage of its own; see, e.g., Controversy 6 in Section 5.7 above.

Skirmish 3: the content of teaching and educational norms

Many constructivisms question the idea of educational norms (Section 6.2.3 above). However, critics in science education take issue with the idea that students "construct their own sciences" during education as per the Piagetian maxim "to understand is to invent". Critics point out that even if existing theories are not always 'correct', education needs to familiarize learners with what the broader intellectual community regards as the products of science, and that it is patently impossible for students to themselves construct ideas that scientists have spent centuries researching (e.g., Phillips, 2000, pp. 14, 32, 179).

A related criticism concerns the constructivist dismissal of standardized assessment:

> *In the hands of the most radical constructivists,* [the dismissal of standard evaluations] *implies that it is impossible to evaluate any educational hypothesis empirically because any such test necessarily requires a commitment to some arbitrary, culturally-determined, set of values.* [. . .] *If the "student as judge" attitude were to dominate education, it would no longer be clear when instruction had failed and when it had succeeded, when it was moving forward and when backward.* [. . .] *To argue for radical constructivism seems to us to engender deep contradictions. Radical constructivists cannot argue for any particular agenda if they deny a consensus as to values. The very act of arguing for a position is to engage in a value-loaded instructional behavior.* (Anderson et al., 2000b)

Skirmish 4: Piaget's legacy and the brand of cognitive science

Radical constructivists and cognitive scientists – and others – both draw heavily on the work of Jean Piaget and hold him in great esteem. This has led to arguments between the groups over whose interpretation of Piaget is correct (or more viable). Some cognitivists advise radical constructivists to construct "a more careful understanding of Piaget" that also acknowledges the critical role of the relatively passive process of assimilating knowledge into existing structures (Anderson et al., 2000b). Radical constructivists consider non-radical understandings of Piaget to be naïve (e.g., von Glasersfeld, 1982, 1995, 1998), and suggest that critics who claim that Piaget failed to give sufficient consideration for social issues ought to apply "the necessary attention" to his original work.

Adding perceived insult to injury is the fact that various constructivist authors state that their work is in line with findings from cognitive science, a claim that Anderson et al. (2000b) – who are eminent cognitive scientists – vehemently deny. Anderson et al. do not cite any examples from computing education, but one suspects that they might consider the book chapter by Greening (1999) (discussed above in Section 6.7) to be a case in point: as Greening pushes a constructivist agenda, he suggests that constructivism contributes to educators' awareness of how important it is to avoid cognitive load, although the critical cognitive scientists are suggesting that it is precisely constructivist teaching practices that are *causing* cognitive overload (see Skirmishes 5 and 6 below). Both views may be right at the same time, in their own ways: even if some constructivist teaching practices tend to cognitively overload students, good constructivist teachers are more likely to use substantial scaffolding that reduces cognitive load (although perhaps still not as much as critics might like; see Skirmish 7).

Skirmish 5: ignoring evidence

Various authors have complained about the vagueness of constructivist ideas. For instance, Prior McCarty and Schwandt (2000, p. 42) describe knowledge construction as "a range of hazily imagined mental activities", while Anderson et al. (2000b) claim that constructivists "refuse to focus on details and precise specifications". Indeed, many constructivists seem to be satisfied with a highly abstract notion of construction, although conceptual change theorists (Section 6.5) and cognitive constructivists who rely on schema theory and mental models are exceptions.[9] The vagueness of instructional frameworks and a lack of rigorous studies and testable hypotheses have been acknowledged as weaknesses also by some scholars otherwise sympathetic to constructivism (Tobias and Duffy, 2009).

A related criticism is that constructivisms are said to be oblivious to much that is known through research. Mayer (2004) and Kirschner et al. (2006) accuse constructivists of ignoring half a century of empirical evidence from cognitive psychology regarding the human cognitive apparatus and its effects on learning. Anderson et al. (2000b) pursue a similar line:

> *This* [constructivist] *criticism of practice (called "drill and kill," as if this phrase constituted empirical evaluation) is prominent in constructivist writings. Nothing flies more in the face of the last 20 years of research than the assertion that practice is bad. All evidence, from the laboratory and from extensive case studies of professionals, indicates that real competence only comes with extensive practice.*

Critics have called for radical constructivists and supporters of constructivist pedagogies to produce testable predictions and empirical data that supports their claims. At least part of the challenge may go unheeded, since some extreme forms of constructivism seem even to deny the possibility or relevance of empirical data to educational decisions (Prior McCarty and Schwandt, 2000; Anderson et al., 2000b).

Skirmish 6: complex, authentic contexts

An issue of disagreement between (some) cognitive psychologists and (some) constructivists is the constructivist claim that knowledge is context-dependent and effective learning requires rich contexts, authentic settings, and ill-structured problems which learners make sense of largely on their own. Mayer (2004), Anderson et al. (2000b) and Kirschner et al. (2006) cite numerous empirical studies whose results indicate that knowledge can be transferred between contexts (under the right conditions), and that ill-structured problems, complex settings and other constructivist pedagogies tend to cognitively overload learners and are commonly ineffective.

The psychologists that I have cited do not claim that authentic contexts are never useful or that no knowledge is context-dependent or best learned in a complex context. Anderson et al., for instance, do agree that some skills are best practiced in a complex setting, for motivational reasons or to learn special skills that are unique to the complex situation. However, they object to the sweeping statements on this topic that many constructivists make.

Mayer (2004) warns against "the constructivist fallacy" of equating behavioral activity (hands-on tasks) with fruitful cognitive activity. According to Anderson et al. (2000b), situated learning and constructivism are movements which claim to oppose behaviorism, but which, through their neglect of cognition, have ironically ended up reiterating Burrhus F. Skinner's utopian vision of behaviorist education from his novel *Walden Two*, in which people learn best from being subjected to the control of the community in which they participate.

Situated learning theory places a particularly heavy emphasis on how effective learning is dependent on participation within a complex, authentic context. This makes situated learning a prime target for the critical cognitive scientists. One of the assertions of Kirschner et al. (2006) is a succinct antithesis of the situated theory of learning: "The practice of a profession is not the same as learning to practice the profession."

[9] A colleague of mine once half-seriously commented that "constructivism can be used to justify any pedagogical innovation" and – on a separate occasion – that "constructivism can't be used to justify anything". This apparent contradiction is explained by the fact that constructivism, when taken in the broad sense, is so abstract when it comes to the processes of learning that it says everything and nothing at the same time.

Skirmish 7: minimal guidance pedagogy

Problem-based learning, inquiry learning, and other constructivist teaching approaches are sometimes labeled *minimal guidance* approaches, as in the title of the provocative paper by Kirschner et al. (2006): *Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching*. Kirschner et al. review evidence that points to the supremacy of direct-guidance methods (such as worked-out examples) in teaching. Influential threads within educational psychology, such as cognitive load theory (Section 4.5), emphasize the need to give significant guidance to novice learners, even to the extent of providing ready-made solutions to problems for the learner to study. This contrasts with the fairly common interpretation of constructivist learning theory according to which learners should be left to discover meanings, patterns, and solutions on their own, with little guidance from a teacher. Such a combination of complex contexts with limited guidance has come in for particularly harsh criticism from the cognitivist camp (see, e.g., Mayer, 2004; Kirschner et al., 2006). Mayer laments that in the face of overwhelming evidence to the contrary, pure discovery pedagogies seem to reappear every decade in a different guise – presently constructivism – "like some zombie that keeps returning from its grave". Sweller has argued that constructivist pedagogy has failed to account for the different kinds of instruction needed to learn *biologically primary knowledge* that we have evolved to acquire easily and automatically – such as language, which can be taught by immersing the learner into complexity – and *biologically secondary knowledge*, such as politics or computer programming, which call for direct, explicit instruction (Sweller, 2010a; Sweller et al., 2007)

The paper by Kirschner et al. (2006) has given rise to a rich exchange of views between academics from the two schools of thought (Schmidt et al., 2007; Hmelo-Silver et al., 2007; Kuhn, 2007; Sweller et al., 2007; Tobias and Duffy, 2009). The typical constructivist defense against the attack of Kirschner et al. has been to emphasize how constructivism – or one's particular interpretation of it, at least – is not a pure discovery or minimal guidance approach. Proponents of various constructivist pedagogies such as PBL have been quick to point out that although some implementations of constructivism might involve little direct guidance, at least the particular variant that is being promoted features significant scaffolding. These arguments have not been entirely to the satisfaction of direct-instruction advocates: disagreement persists concerning the amount of guidance needed, and in particular on whether providing the solution to a problem is an inappropriate form of strong scaffolding or a very useful one. Many constructivists emphasize that methods such as PBL improve 'softer' skills that the attacking psychologists do not test for, such as teamwork and self-directed learning skills. Some (such as Kuhn, 2007) even question the goal of teaching knowledge, preferring to emphasize the development of generic thinking and learning skills (which is a goal that goes against the schema theory view of expertise as domain-knowledge dependent; Chapter 4).

Both sides of the argument have sought to present empirical evidence for their respective views and criticized the studies cited by the other side. The book edited by Tobias and Duffy (2009) features some increasingly fruitful dialogue between the two camps. Future research may help us gain better understanding of under what circumstances and for which goals the different pedagogies work best, and how they can complement each other.

Skirmish 8: nothing new

Nearly all critics of constructivism, it seems, have cast doubt upon the originality of constructivist ideas and the pedagogical practices they result in. Phillips (2000), for instance, asks if constructivism has really changed anything or if it is just the newest form of Kant's philosophy, Dewey's progressivism, and discovery learning. Matthews (2000) suggests that constructivism has mostly succeeded in introducing new jargon that – instead of simplifying complex matters, as terminology should – complicates simple matters. Matthews even provides a tongue-in-cheek dictionary that maps "constructivist new speak" to "orthodox old speak". Elsewhere, Matthews (1992) calls constructivism old wine in new bottles, while Levitt (quoted by Patton, 2002, p. 101) goes one better as he describes constructivism as a manifestation

of postmodernism that is essentially "a particular technique for getting drunk on one's own words".[10]

The less extreme and more wishy-washy one's constructivism is, the more susceptible one is to accusations of unoriginality and weak potential for genuine change. Greening (1999) warns us that constructivism cannot be merely approached, it must be accepted in its entirety, lest we end up merely diluting our curricula with an assortment of constructivist principles that mean little in isolation from each other. According to Larochelle and Bednarz (1998), "softer" non-epistemological versions of constructivism have "scarcely modified the usual teaching modus vivendi at any level of instruction one chooses to examine". Some critics of constructivism agree with this latter claim: "A less radical constructivism may contain no contradictions and may bear some truth. However, [...] such a moderate constructivism contains little that is new and ignores a lot that is already known" (Anderson et al., 2000b).

Skirmish 9: a disconnect between pedagogy and theory

A corollary of the claim that constructivist theory is vague (see Skirmish 5 above) is that there is a disconnect between constructivist theory and the pedagogies that are commonly advocated by constructivists. That is, it "seems possible for a person who accepts constructivism as a philosophy to adopt any variety of pedagogical practices (or for a teacher who uses constructivist classroom practices to justify doing so in a variety of ways, some of which might not philosophically be constructivist at all)" (Phillips, 2000, p. 18). Many constructivists and critics have commented on this issue; some see it as a problem, others do not.

Ben-Ari (2001a) asks the delicate question of whether "being a constructivist requires an epistemological commitment to empiricism and idealism (or social idealism), as opposed to rationalism and realism that seem to come more naturally to scientists". Citing the work of Matthews, Ben-Ari further notes how "pedagogical constructivists" avoid the problem by concentrating solely on improving pedagogy and ignoring epistemological details as not worth disputing. The question then becomes whether we need constructivist theory at all. Prior McCarty and Schwandt (2000) certainly do not think so, arguing that even though some constructivists may advocate good pedagogical practices, those practices hardly require, or benefit from, justification in the form of the "garage sale of outdated philosophical falsisms" that is constructivist theory. An example of an explicitly constructivist epistemological position which is nevertheless based on "traditional" pedagogy and experimental findings from cognitive psychology is that of Renkl (2009).

I give the last word to Greening (1999), who points out that one may care less "whether constructivism is *required* to bring about the beneficial changes in learning that have been attributed to it, and more about whether or not it *has*" (p. 50, emphases in original). Even a skeptic of constructivist theories accepts that the constructivist movement is influentially advocating certain pedagogies. Because of this, if for no other reason, it is important to consider how present-day pedagogical innovations relate to constructivism.

[10] Adding to this curious theme linking constructivism and alcohol, Richards's (1995) description of the problems of constructivism is zymological: social and personal constructivism are analogous with two non-communicating descriptions of the process of beer-brewing.

Chapter 7

Phenomenographers Say: Learning Changes Our Ways of Perceiving Particular Content

Phenomenography is a primarily qualitative tradition of empirical research, whose main contributions lie within education and its hybrid disciplines such as CER. Since the 1970s, phenomenographers have sought to explore the educationally critical variation in how people experience, perceive, or understand various phenomena.[1] A phenomenon of interest can be specific – such as number in kindergarten mathematics, matter in physics, or variable in computing – or more generic – such as learning to program or even learning in general. Learning, from the phenomenographic point of view, involves becoming able to experience phenomena in significantly different new ways.

In this chapter, I introduce the phenomenographic research tradition and some of its epistemological and learning-theoretical foundations. This enables us to then review phenomenographic work on programming education. My treatment of phenomenography in this chapter will be extended by Chapter 17 in connection with a phenomenographic study on visual program simulation.

Section 7.1 outlines the constitutionalist belief system which provides an epistemological backdrop to phenomenographic research. In Section 7.2 we get a glimpse of the kinds of results that phenomenographic research produces. The implications of phenomenographic theory and research for learning are the topic of Section 7.3. Pedagogical issues follow in Section 7.4, and bring us back to the focus of Part II – what it takes to learn programming – which I discuss from a phenomenographic perspective in Section 7.5. The chapter concludes with the consideration of some criticisms of phenomenography in Section 7.6.

7.1 Phenomenography is based on a constitutionalist worldview

> *Our world is a world which is always understood in one way or in another, it can not be defined without someone defining it. On the other hand, we can not be without our world.* (Marton, 2000, p. 115)

Phenomenographic research is associated with a worldview that is sometimes termed *constitutionalism* (Prosser and Trigwell, 1999). Constitutionalism, which draws on the phenomeno*logy* of Edmund Husserl[2], has been presented as a basis for phenomenographic research by Marton (2000; Marton and Booth 1997). From the constitutionalist point of view, the only world that humans can meaningfully think about is their 'lifeworld' of lived experience. Understanding exists as an internal, two-way relationship between human experiencers and the phenomenon that is experienced. It is inseparable from the experiencer and phenomenon alike, and is a reflection of both.

[1] The words "experience", "perceive", and "understand" are often used interchangeably in the phenomenographic literature. I do the same in this chapter. Phenomenographers use these words in a particular sense. The intended meaning should become clearer over the course of the chapter, and further in Chapter 17.

[2] For linkages between phenomenography and (Husserl's version of) phenomenology, see, e.g., Marton and Booth (1997) and Uljens (1996).

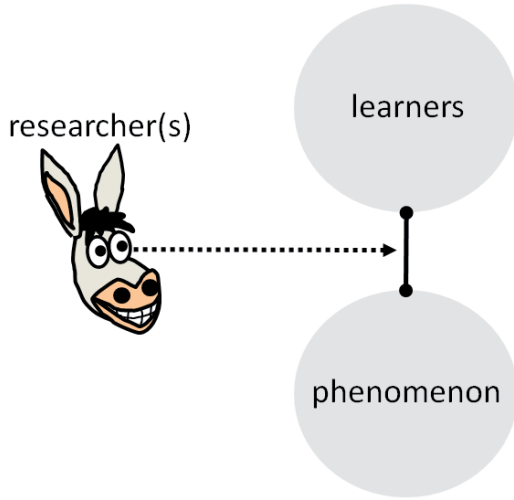


Figure 7.1: The phenomenographic research perspective. The researcher investigates the relationship between a population and a phenomenon; the results are a reflection of both. I have followed a fledgling tradition sparked by Berglund (2002) in my asinine portrayal of the phenomenographer in this figure.

The constitutionalist view is in sharp contrast to cognitive psychology (Chapters 4 and 5) and many forms of constructivism (Chapter 6), as it discards the notion of mental representation, viz., the notion that there exist representations within minds that are distinct from the phenomena that they represent. Marton (2000), for instance, rejects the idea that experiences are structures in a separate mental world, accusing cognitive psychologists of an untenable mind–matter dualism (see Section 5.7). By adopting a view of knowledge as ways of experiencing the world – as internal relations between knower and known – we achieve, Marton argues, a paradox-free non-dualist epistemology. Uljens (1998) remarks that Marton's "non-dualism" is essentially a type of monism. There is only one world, within which people, phenomena, and their relationships exist alike.

Philosophical considerations aside, two practical points arise from the constitutionalist stance. The first concerns research: a phenomenographer investigates neither people nor phenomena exclusively, but their relationship, which reflects both at once. The second concerns learning: learning must be considered in terms of both the learner and the specific content (phenomenon) being learned about. Let us first look at the nature of phenomenographic research and the type of results it produces, and return to learning in Section 7.3.

Investigating experiences of phenomena

> *We should explore* [...] *what the world we experience is like, on the one hand, and what our way of experiencing the world is like, on the other hand. And of course: these are not two things. They are one.* (Marton, 2000, p. 115)

Phenomenographers seek to find out how people experience phenomena. But what exactly is a phenomenon and what does it mean to experience one? The constitutionalist answer is that these two questions are inextricably linked. When he investigates experience, the phenomenographic researcher studies the relationships between people and phenomena (Figure 7.1).

A phenomenon is "something in the world – concrete or abstract – which can be delimited from the world, by the researcher and by others, according to their knowledge of the world" (Booth, 1992, p. 53). Marton (2000) defines a phenomenon as a complex of all the different ways in which something can be experienced. A way of experiencing something is one of the various facets that together constitute

the phenomenon. An individual's experience of something is shaped by both the experiencer and the phenomenon.

An example. Marton and Booth (1997) summarize a phenomenographic study by Marton, Watkins, and Tang, which explored Open University students' experiences of what learning is. Learning was experienced in one of six ways: increasing one's knowledge, memorizing and reproducing, applying knowledge, understanding and gaining insights about the world, opening one's mind to see things in a different way, and changing oneself as a person. These ways of experiencing learning play a part in constituting what the phenomenon of learning is (along with other ways of experiencing learning that the particular study did not discover). The research results speak not only of the students, or of what learning is, but of both.

7.2 There is a small number of qualitatively different ways of experiencing a phenomenon

People experience the same phenomenon differently. Even the same person experiences the same phenomenon differently at different times and in different contexts. Are there countless significantly different ways of experiencing the same phenomenon? Phenomenographers posit that it is not so.

> *The constitutionalist view differs significantly from constructivism in that learners are seen to experience what they are learning in a small, identifiable range of different ways (usually between three and seven). An identifiable range of variation is thus assumed to be present in any given group (as compared with the idiosyncratic construction of every individual).* (Bruce and McMahon, 2002, p. 12)

This is not to deny that everyone's conception of a phenomenon is unique. However, it is posited that a researcher can describe ways of experiencing a phenomenon in an abstract way that captures the telling differences between the few qualitatively different ways of experiencing the phenomenon.

Critical aspects

A phenomenon, viewed from a particular perspective, is characterized by certain *critical aspects* (also known as *critical features*; I use these terms interchangeably). These aspects are critical in that they define how ways of perceiving the phenomenon are qualitatively different:

> *Every phenomenon that we encounter is infinitely complex, but for every phenomenon there is a limited number of critical features that distinguish the phenomenon from other phenomena. What critical features the learner discerns and focuses on simultaneously characterises a specific way of experiencing that phenomenon.* (Pang, 2003, p. 148)

Let us take an example from CER. Eckerdal (2006) investigated how CS1 students understand what an object is. Her data – typically for a phenomenographic study – consisted of in-depth interviews with learners. Eckerdal describes three qualitatively different ways of understanding the phenomenon of object, which I have summarized in Table 7.1. There is a logical structure to the three categories listed in the table. Category A describes a very simple way of understanding what an object is: a piece of code. Only the critical aspect of program text is in focus. In the understanding described by Category B, program code is also recognized as a defining aspect of objects, but objects are additionally seen as active entities during a program run. This relationship between objects and execution-time events is another critical aspect of objects. The understanding in Category C includes those from the two first categories, and further attributes a world-modeling aspect to objects.

Outcome spaces and categories of description

The results of a phenomenographic study usually take the form of an *outcome space* that consists of a small number of interrelated *categories of description*. Table 7.1 is an example of an outcome space

Table 7.1: Qualitatively different ways of understanding the programming concept of object (adapted from Eckerdal, 2006).

Code	Description
A	An object is experienced as a piece of code.
B	An object is experienced as a piece of code, and as something that is active in the program.
C	An object is experienced as a piece of code, as something that is active in the program, and as a model of some real-world phenomenon.

with three categories of description. In each category, the researcher has crystallized a description of a particular way of understanding that is different from the others in an educationally critical sense.

A category does not represent a particular person's understanding or mental representation of a phenomenon, although it is possible for a person, at a particular time, to conceive of the phenomenon in a way that is consistent with a category. The categories are not defined in isolation; an outcome space is defined by the way the conceptions contrast with each other. These differences between categories arise out of the way certain critical features – or relationships between features – are discerned in one category but not in another.

This hierarchicality of Eckerdal's outcome space is typical of phenomenographic results, as is the way some types of understanding can be considered richer while others are more limited. Compared to a richer understanding of a phenomenon (e.g., one that matches Category C), a limited understanding of a phenomenon (e.g., one that matches Category A or B) either focuses on a subset of the critical aspects of a phenomenon or fails to discern relationships between particular critical aspects.

We now have an inkling of what phenomenography is and how phenomenographers view human experience. What does all this mean for learning?

7.3 Learning involves qualitative changes in perception

> *Knowing the phenomenon can be seen as having a multi-faceted view of the* [phenomenal] *object, while learning about it is gaining access to views of further faces and developing an intuitive relationship with the object so that an appropriate face or set of faces is seen in appropriate circumstances* (Booth, 1992, p. 261)

7.3.1 Discerning new critical features leads to learning

Most phenomenographic work focuses on a certain kind of learning that takes place in educational settings: learning to experience particular phenomena in richer, significantly different ways than before, in keeping with learning goals set by teachers or the learners themselves. The discourse on learning within the phenomenographic tradition may sound limited – after all, learners should learn to *do* things, they should acquire new skills, not just new understandings. However, phenomenographers are in fact very concerned with concrete abilities, which are seen as an outcome of learning to experience phenomena in richer ways. "If we are able to handle a situation in a more powerful way, we must first *see* it in a powerful way, that is discern its critical features and then take those aspects into account by integrating them together into our thinking simultaneously, thus seeing them holistically" (Marton, 2007, p. 20). By discerning what is critical about that which they encounter – and what is not critical, but a mere detail – learners become prepared to deal with future situations, which they can make sense of in terms of their critical aspects

(Bowden and Marton, 2004). A way of experiencing is the key to a new way of doing.[3]

From the phenomenographic perspective, learning is not a process that happens solely within a learner's mind, nor something external. To learn to experience the world in a significantly different way implies a change in the two-way relationship between the learner and a phenomenon within the world that they both inhabit. Such changes bring about new ways of discerning the meaning of the phenomenon, the part–whole relationships within the phenomenon, and the relationships of the phenomenon to its surroundings (Marton and Booth, 1997).

The relational view of learning emphasizes that learning is not generic, but involves the development of new perspectives on particular phenomena. As learning always involves both a learner and a phenomenon, it is not sufficient to consider merely cognitive processes that enable learning to take place, but also the content of learning, the particular features of the phenomenon that characterize the significantly different ways of understanding it.

7.3.2 The discernment of critical features requires variation

Pang (2003) distinguishes between two "faces of variation" that are relevant to phenomenographic research. The first type of variation exists in the differences between qualitatively different ways of understanding a phenomenon. This is the kind of variation that I have discussed so far in this chapter. Another kind of variation pertains to a particular critical feature of a phenomenon. This variation is key to discerning a critical feature in a particular way and, consequently, to learning to experience a phenomenon in a significantly different way. The second kind of variation is the domain of variation theory.

Variation theory (Marton and Tsui, 2004; Pang, 2003) is a theory of learning closely associated with, and inspired by, constitutionalist epistemology and the phenomenographic research movement. Its chief tenet is that in order to learn, the learner must discern variation in educationally critical features of the *object of learning*, that is, what one learns about.

As discussed above, to learn to experience a phenomenon in a new way requires new critical features, or relationships between critical features, to be discerned. According to variation theory, each critical feature is associated with a *dimension of variation*. To be able to discern a feature, one must experience variation along the dimension of variation corresponding to that feature.

The jargon of variation theory can get complicated, but the basic idea is very simple, indeed quite commonsensical. To experience something as red, we must experience other colors as well. To experience the height of a person, we must experience people of different heights. Different colors and different heights are values along the dimensions of variation in color and height, respectively. A strategy for solving a programming problem is a value along a dimension of variation; other strategies for solving similar problems are values along the same dimension. To compare strategies, you have to have experienced more than one.

To experience variation in a dimension requires simultaneous awareness of different values along that dimension, either through being exposed to the values at the same time or by juxtaposing one's current experience with prior experiences. In addition, the learner must be able to experience each of the critical features as features of a single phenomenon, joined together in his experience. That is, the learner needs to be focally aware of each critical feature simultaneously so that the critical features are present at the same time as features of the phenomenon, with relationships to other critical features and to the phenomenon as a whole (Marton et al., 2004; Pang, 2003).

7.4 Phenomenographers emphasize the role of content in instructional design

To a teacher, a phenomenographic outcome space may be useful as a tool for analyzing their own teaching, the content they teach about, and the ways in which their students understand (or might understand) the

[3] This is not to say that this kind of learning is the only one there is. Phenomenographers acknowledge that there is learning that involves (merely) further developing or fine-tuning a perspective – a way of experiencing – which one already has access to. However, it is the presumably more profound kind of learning brought about by qualitative shifts in experience that phenomenography focuses on.

content. "The assumption is that if we want to make the student think in a certain way about something, it should be useful to know what other ways there are to think about it" (Johansson et al., 1985, p. 255). Marton and Booth (1997, p. 81) suggest that phenomenographic outcome spaces serve to identify "a notional path of developmental foci for instruction". Some phenomenographers use variation theory as an additional tool to analyze and report differences in ways of experiencing, and to offer pedagogical recommendations.

7.4.1 Teachers' job is to aid students in discerning critical features

In many cases, it is not enough for a teacher to just say to students what the critical features of a phenomenon are. In order for students to actually experience these features, the teacher needs to help learners discern variation in the corresponding dimensions, wherever those dimensions are not already familiar to the learners.

Marton et al. (2004) discuss variation-based pedagogy in terms of creating a *space of learning*, which is "the pattern of variation inherent in a situation" and "comprises any number of dimensions of variation and denotes the aspects of a situation, or the phenomena embedded in that situation, that can be discerned due to the variation present in the situation". A space of learning "delimits what can be possibly learned (in sense of discerning) in that particular situation" as it is "a necessary condition for the learner's experience of that pattern of variation unless the learner can experience that pattern due to what she has encountered in the past" (p. 21).

Marton et al. are at pains to emphasize that their work is about 'making learning possible' rather than guaranteeing learning results, as "no conditions of learning ever *cause* learning" (p. 22). That is not to say that they place the burden of learning entirely on the shoulders of the learner; on the contrary, the teacher's expertise plays a great role in constructing an effective space of learning. The job of the teacher is to facilitate discernment by creating situations in which students get to experience the critical variation. Learners must have the opportunity to observe values along the dimensions of variation that correspond to the critical features of the object of learning. For instance, when learning about object-oriented programming, students should be presented with opportunities to observe objects as code, as interacting agents within programs, and as parts of domain models. Without such a space of learning, Marton et al. argue, a qualitative shift in perception is not possible. Eventually, the critical features of the phenomenon should be discerned not only in isolation but as interconnected aspects of the object of learning; such discernment, too, is affected by how the phenomenon is present in the space of learning.[4]

It is not only what varies that is relevant, but also the static context against which the variation appears and against which it can be discerned. Marton et al. (2004) encourage teachers to think about and exploit 'patterns of variation' through texts, speech, and learning materials to enable learners to discern new variation. These patterns include contrast (comparing values along a dimension of variation), generalization (showcasing an aspect by demonstrating different instances in which it features), separation (varying an aspect while others remain invariant), and – often the most challenging – fusion (varying multiple features at the same time to appreciate their relationships and give a holistic 'feel' for the phenomenon).

7.4.2 For critical features to be addressed, they should be identified

Content-based pedagogy

Constructivists (Chapter 6) promote active and collaborative learning methods such as problem-based learning and groupwork. Cognitive load theory (Section 4.5) emphasizes the usefulness of managed assignments such as worked-out examples. What teaching methods do phenomenography and variation theory recommend?

[4] Ko and Marton (2004, p. 62) describe good mathematics teaching as "planned, choreographed, and well thought-out lessons" which nevertheless offer "plenty of space for the students' own independent and spontaneous ideas". In relation to the present-day buzzwords "teacher-centered" and "student-centered", Ko and Marton note, good teaching can be equally both, if "teacher-centered" is taken to mean that the teacher has the key role of making sure that a space of learning is created that matches the intended learning outcomes, and "student-centered" is taken to mean that students take ownership of the space of learning and participate in creating it.

The answer is none in particular, which is to say, none in general (see, e.g., Marton and Booth, 1997; Marton and Tsui, 2004; Bowden and Marton, 2004). Phenomenographers have contended that many teachers do not evaluate pedagogical approaches in terms of how well they are suited to dealing with particular content, and that many scholars likewise underplay the role of content, chasing instead the chimeric "art of teaching all things to all men". Marton and colleagues oppose the notion of a general pedagogical aid, be that project-based learning, example-based direct instruction, peer teaching, or IT-aided learning. Although these often-proffered techniques may be very useful for some content and in some circumstances,

> *no general approach to instruction can ever ensure that the specific conditions necessary for the learning of specific objects of learning are brought about. In order to do this, we must take the specific objects of learning as our point of departure.* (Marton and Tsui, 2004, p. 229)

What phenomenographers recommend is to use whichever method works for highlighting the particular variation needed in order to learn about the particular content that is being taught (when that content is considered from the point of view of what the learners are intended to learn about it). What works must be discovered for each object of learning separately and is tied to the educationally critical features of the object.

Teachers or researchers?

Pedagogic recommendations based on phenomenography and variation theory blur the line between the teacher and the educational researcher.

If pedagogic solutions need to be discovered for each object of learning separately, if perhaps even generic capabilities such as thinking strategies and communicative skills are domain-specific (as suggested by Marton and Tsui, 2004, p. 229), then the good teacher must be very aware of the pedagogically sensitive aspects of the content they teach. Variation theorists call for teachers to engage "in finding out what the specific conditions [of learning] are in every specific case, and how they can be brought about" (ibid., p. 231). This is a non-trivial task even for the content expert, Bowden and Marton (2004) contend, as the more fundamental and important one's way of seeing things is, the harder it is to even notice it.

In the light of variation theory, teachers should define objects of learning in terms of ways of experiencing particular content, learn about the ways in which learners may experience that content, and determine what variation is critical for experiencing the content in the intended kind of way. Such explorations enable the teacher to enact learning situations which afford the educationally critical variation, and which have a motivating *relevance structure* (Marton and Booth, 1997, Chapter 8) that draws students to experience the critical variation.

From a phenomenographic point of view, the goals of the teacher, then, have much in common with those of the phenomenographic researcher who studies the relationships between people and particular phenomena. Teachers should build on research findings that are particular to the object of learning they wish to teach about.

With that in mind, let us look at what phenomenographic research says about introductory programming.

7.5 Learning to program involves qualitative changes in experience

Phenomenography has gained popularity in the past decades, a development which has impacted on CER as well as other fields. There is a small but growing body of phenomenographic work that investigates the challenges of learning to program. This work can be roughly sorted into two threads. The first thread is concerned with very general questions: in what ways do learners experience what programming is and what it means to learn to program? The second thread is more varied and investigates the ways in which particular programming concepts are understood. Both of these threads were influenced by the seminal doctoral thesis by Booth (1992), which is a good place for us to start looking.

Table 7.2: Different ways of experiencing learning to program (adapted from Booth, 1992).

Code	Description
A	Learning to program is experienced as learning a programming language (or several).
B	Learning to program is experienced as above, and as learning to write programs using various techniques and language features.
C	Learning to program is experienced as above, and as learning to solve problems in the form of programs.
D	Learning to program is experienced as above, and as becoming participant within a programming community.

7.5.1 Learners experience programming differently

Booth (1992) conducted a pioneering phenomenographic study of learning introductory programming, based on interviews with novice programmers during a semester-long CS1 course. She found that her interviewees experienced *what it means to learn to program* in four qualitatively different ways. This outcome space, which I have paraphrased in Table 7.2, features a hierarchical progression from a rudimentary way of understanding ('just learning a language') in Category A to the rich way of understanding in Category D ('learning to solve problems as part of a community'). Booth also found that her interviewees had three qualitatively different *ways of experiencing what programming is*: A) something directed towards the computer; B) something directed towards a problem that one intends to solve, and C) something directed towards a product that is the outcome of the programming activity.[5] *Programming languages*, Booth reports, were experienced in four different ways: A) as utility programs that belong in computer system; B) as sets of code elements of which programs are built; C) as a means of communicating between program components, the programmer, and/or a program's users, and D) as a medium of expression that allows the programmer to express solutions to problems.

Later studies have developed the themes opened up by Booth (1992), lending support to and complementing her findings.

Booth (2001b) herself takes her earlier work – in which the richest ways of understanding are characterized by a communal aspect (e.g., Table 7.2) – and relates it to situated learning theory (Section 6.6) in order to explore the relationships between her categories and three computing subcultures: the academic, the professional, and the informal culture.

Bruce and her colleagues also studied learners' experiences of learning to program (see, e.g., Bruce et al., 2004; Stoodley et al., 2004). Bruce et al. (2004) report that their interviewees experienced *learning to program* as A) following the structure of a programming course to keep up with set assignments; B) learning to code using the right syntax; C) learning to write programs through understanding and integrating the concepts one encounters; D) learning to do what it takes to solve problems, and E) learning what it takes to be a part of the programming community and to think like a programmer. The findings of Bruce et al. partially match and partially extend Booth's earlier work.

Eckerdal et al. (2005) observed that students talk about learning to program in terms of developing a special "way of thinking", which is different from everyday thinking and from the thinking in other subjects they had learned about. Eckerdal et al. report an interview-based phenomenographic study in

[5] It is perhaps useful at this point to clarify that the phenomenographer seeks to look beyond language. Booth's primary focus, for example, was not on the different meanings that students assigned to the *word* "programming" but on the different ways in which students related to the subject they studied. The meaning(s) of the word "programming" are a separate (even though related) issue that is also an important consideration for the programming teacher (cf. Walker, 2011) and indeed for the phenomenographic analyst who tries to get at the underlying conceptualizations.

Table 7.3: Different ways of experiencing programming (adapted from Thuné and Eckerdal, 2010)

Code	Description
A	Computer programming is experienced as using a programming language for writing program texts.
B	Computer programming is experienced as above, and as a way of thinking that relates instructions in the programming language to what will happen when the program is executed.
C	Computer programming is experienced as above, and as producing applications of the kind familiar from everyday life.
D	Computer programming is experienced as above, and as a 'method' of reasoning that enables problems to be solved.
E	Computer programming is experienced as above, and as a skill that can be used outside the programming course, and for other purposes than computer programming.

which they found that students experience *learning to program* in five different, hierarchically connected ways: A) as learning to understand and use a language; B) as learning a hard-to-define way of thinking related to a programming language; C) as learning to understand computer programs in real life; D) as learning a 'method' of thinking which enables problems to be solved, and E) as learning a skill that can be used outside the programming course. Thuné and Eckerdal (2009, 2010) analyzed students' experiences of *what programming is*. They describe a hierarchical outcome space (Table 7.3), whose categories largely mirror those from Eckerdal et al.'s results – as one might expect, since conceptions of programming and learning to program are interdependent.

7.5.2 A limited way of experiencing programming means limited learning opportunities

Some of the above ways of understanding programming and learning to program are very simple and do not provide a fruitful basis for learning programming concepts. For instance, a learner who perceives learning to program merely as learning to write program text according to syntactic rules, will inevitably miss out on many opportunities to learn until his overarching view of programming changes in a significant way.

The outcome spaces produced by phenomenographic studies within CER suggest that learning to program may require multiple large-scale qualitative shifts in learners' perceptions of programming. A static perspective limited to program text needs to be developed into a dynamic one that takes execution-time behavior into account. Ultimately, one should learn to view programming as an empowering skill applicable to problems in one's community.

Pedagogical implications

In line with variation theory, phenomenographers in CER emphasize the importance of highlighting variation in the critical aspects of phenomena, rather than championing any specific teaching methods as blanket solutions. When teaching methods such as groupwork and visual tools are advocated, it is with the caveat that these pedagogies will only be useful if they help learners discern new critical aspects and relationships between them. Unsupervised groupwork, for example, may, but also may not, lead to discernment of critical variation. Often, the role of the teacher as a facilitator and guide rather than a transmitter of information, is emphasized. Here is the view of Booth (1992), for instance:

Teaching should encourage well-founded conceptions through example. Teaching should not merely try to bring about expert behaviour in students by offering expert views on the content, but give students the range of challenges which enable them to come to the expert understanding via experience. Teachers should above all be aware of the range of conceptions held by their students, and that poor conceptions are not necessarily caught in lab exercises and examinations.

To learn to view programming in terms of execution-time behavior, for example, learners need to be placed in environments that enable and motivate them to become focally aware of the execution-time behavior of programs and its relationship with program code. Thuné and Eckerdal (2009) apply variation theory to their outcome space concerning conceptions of programming (Table 7.3). They recommend that instruction bring students to focus not only on various pieces of program code, but also: different program actions during execution, different applications of programming to everyday life, different problems for which there are programming solutions, and different contexts within which programming skills can be used. They further warn against varying too many aspects at once and provide examples of teaching techniques that, it is hypothesized, highlight the critical aspects. For instance, making students aware of how a tiny change in program code can lead to dramatic changes in program behavior may be effective in helping students pay attention to the runtime aspect of programs. After this aspect is discerned, software tools may be used to illustrate the relationship between program code and the resulting behavior.

As it is important for a teacher to be aware of the qualitatively different ways in which learners may understand the subject matter, the pedagogical expertise of teachers and teaching assistants is highly important. Booth (2001a) reports a study in which it turned out that the classroom tutors employed did not have a sophisticated way of understanding the goals of the course and in particular failed to see how the content of the course was intended to relate to a broader context. Matters improved greatly with tutor training: "An important insight the course team has gained is that the insights gained by the students can hardly be expected to surpass those of the tutor team. The relevance structure for the students can only be brought about through [the] tutor's experienced relevance." (Booth, 2001a, p. 185)

7.5.3 Learners experience programming concepts in different ways

Some phenomenographic projects have studied how programming students experience particular programming topics that are relevant to at least some CS1 courses.

Booth (1992) found several different ways of experiencing the concepts of recursion and function; the concept of *recursion*, for instance, was experienced in three different ways: A) as a construct in a programming language B) as a means of bringing about repetition, and C) conceptually as self-reference that enables a function to make use of itself.

I have already used as an example Eckerdal and Thuné's (2005) outcome space of different ways of understanding what an object is (Section 7.1). The same paper by Eckerdal and Thuné also reports an analogous outcome space for the concept of class. Later work has suggested that students may initially develop either a static "text conception" or a dynamic "action conception" of an object which they then need to relate to the other conception (Eckerdal et al., 2011). I myself have conducted phenomenographic studies of students' ways of understanding what it means to store objects in memory (Sorva, 2007), and of different kinds of variables (Sorva, 2008).

Stamouli and Huggard (2006) studied students' perceptions of program correctness (see also Booth, 1992). Boustedt explored students' understandings of the concepts of interface and plugin (Boustedt, 2009) and UML class diagrams (Boustedt, 2012). Lönnberg (2012) focused on students' perceptions of bugs in concurrent programs. Thompson (2008, 2010) studied practitioner perceptions of object-oriented programming with a view to improving education on the topic.

These studies contribute to the large body of evidence that shows that novices understand many fundamental programming constructs in limited ways that may hinder further learning and are non-viable for many common programming tasks. The lists of 'misconceptions' in Appendix A include many of the specific limited understandings that these phenomenographic studies have uncovered.

7.6 Phenomenography has not escaped criticism, either

In this section, I paraphrase a few main criticisms of phenomenography from the literature and phenomenographers' responses to those criticisms.

Criticism 1: language and the nature of outcome spaces

Critics from both within the phenomenographic movement (e.g., Säljö, 1994) and outside (e.g., Webb, 1997) have criticized the perceived disregard of interview-based phenomenography for the role of language, communication, and social construction. Scullion (2002, pp. 101–103) reviews this debate. According to him, phenomenographers in Marton's tradition see language as a relatively unproblematic device that the phenomenographer must get past to learn about the interviewee's conceptions. Critics such as Säljö call for a more careful and explicit treatment of language:

> *To me, phenomenography has a weak spot in its lack of a theory of language and communication, and in its almost dogmatic disregard for paying attention to why people talk the way they do. The assumption seems to be that what is meant by what is said can be construed as representing a conception of the phenomenon which one – according to the interviewer – is talking about.* (Säljö, 1994)

A related question is what phenomenographic outcome spaces really tell us. Some critics have argued that it is impossible to genuinely explore and describe what other people really experience. For instance, according to Richardson (1999), phenomenographers fail to deliver on their promise to describe the world as people experience it: "they have to depend on other people's discursive accounts of their experience [and] are merely describing the world as people describe it."

Some phenomenographers are happy to agree that outcome spaces are no more than descriptions of how people describe things in a particular kind of situation. This does not mean they are not useful in practice:

> *I don't wish to assert that I 'know' an individual's conception of a phenomenon. What I do want to be able to say is that, following a given interview context, analysis of the transcripts enables me to differentiate between a number of different ways of seeing the phenomenon that are apparent in that kind of conversation.* [...] *Also, it is not possible for the researcher to 'be' that person; the researcher interprets the communication with the person.* [...] *I am satisfied that phenomenographic research produces descriptions which owe their content both to the relation between the individuals and the phenomenon (that is, their conceptions) and also to the nature of the conversation between the researcher and each individual and its context (which includes the relation between the researcher and the phenomenon).* (Bowden, 2000, pp. 16–17)

Bowden's pragmatic position seems a reasonable way to address criticism of this kind.

Criticism 2: value-loaded norms

Webb (1997), who writes from a post-modernist perspective, is critical of the way phenomenography posits that some ways of understanding are qualitatively better or more correct than others. Webb calls phenomenographic research hopelessly value-loaded and contaminated by the researcher's own conceptual apparatus. He accuses phenomenographers of making normative, even dogmatic judgments, and of forbidding learners to develop alternative views.

Phenomenography does indeed posit that some ways of understanding are richer and better than others in the sense that they enable learners to perform more effectively. Marton and Booth (1997, p. 2), for instance, explicitly take a stand:

> *We are living in an age of relativism, but a fundamental principle we are assuming in this book is that education has norms – norms of what those undergoing education should be learning, and what the outcomes of that learning should be.*

If one dismisses educational norms and embraces relativism full on, phenomenography may have little or nothing to contribute; the usefulness of educational norms is probably axiomatic to most phenomenographers.

In introductory programming education perhaps even more than in some other fields, educational norms are needed in order to succeed, as the computer represents an artificial, non-negotiable ontological reality (see Section 6.7.2). Even where acceptable alternative views can be developed, it is useful for university students to learn to 'see' phenomena in the ways that experts presently 'see' them.

Criticism 3: the quality of qualitative research

Phenomenography is, by and large, a qualitative research approach, and many general criticisms of qualitative research also apply to phenomenography. These include doubts about the reliability, objectivity, generalizability, and overall trustworthiness of interpretive research. I will return to these issues in Chapter 16 when discussing the quality of some of my own research. For now, it must suffice to point out that the rigor of qualitative research is typically evaluated differently than that of traditional quantitative research. Many qualitative researchers do not even attempt to fulfill all the goals of traditional quantitative research, such as replicability.

Criticism 4: unenlightening results

Another of Webb's (1997) criticisms concerns the predictability of phenomenographic results, suggesting they tend to tell us little that is new.

> [Phenomenographic activities are] *informed by theory and prejudice. It seems likely then that phenomenographic research will tend to report the history of a particular discipline as it is understood by the researchers and as they reconstruct it through the people they interview. Phenomenographic explanation is prone to reproduction of the discourses it studies.* [...] *A phenomenographer considers himself to have constructed* [the different ways of understanding he reports] *from raw data. It is only of passing interest that these conceptions have a close similarity with the history of the discipline. A critic might ask how it could be otherwise.* (Webb, 1997, pp. 196–197)

Phenomenographers argue back that it is not assumed that the researcher's own understanding does not affect the results, but that rigorous steps are taken to 'bracket' the researcher's understanding of the phenomenon and be as open-minded as possible. Merely reproducing known wisdom is a risk in phenomenography (as well as in qualitative research in general and, indeed, research in general). However, there are clear examples of phenomenography informing education by challenging the existing consensus of the mainstream (an example is Neuman's study of how children experience numbers, described in Marton and Booth, 1997, pp. 57–67).

Criticism 5: doomed to repeat SOLO?

Many phenomenographic outcome spaces bear a resemblance to the SOLO taxonomy (Section 2.2): the lowest-ranking categories describe unistructural ways of understanding that focus on a single aspect of the phenomenon, and richer categories describe increasingly complex multistructural ways of understanding that lead to a coherent relational way of understanding. The outcome space is sometimes capped off by an extended abstract understanding in which the experiencer is capable of taking the phenomenon into novel contexts. Table 7.3 is an example of an outcome space of this kind. This begs the question: are phenomenographic studies doomed to do little more than replicate the SOLO taxonomy in different contexts?

The counterargument is that even when phenomenographic outcome spaces do resemble SOLO, they still describe evidence-supported, concrete instantiations of SOLO for particular phenomena, and detail what particular dimensions of variation pertain to the unistructural, multistructural and relational ways of understanding those phenomena. Prosser and Trigwell (1999, p. 120) point out that SOLO – as its name implies – is concerned merely with the general structure of learning outcomes, whereas phenomenographic

outcome spaces describe both the structure of learners' understandings and the meanings learners assign to particular phenomena (see Section 7.1). This concrete information is helpful for teachers as an analytical tool and helps provide "a notional path of developmental foci for instruction" (Marton and Booth, 1997, p. 81). Taking the SOLO taxonomy to different contexts in a rigorous, empirically founded way is not an unworthy undertaking. In this sense, phenomenographic outcome spaces are considerably more than what you get by 'simply' applying SOLO to a phenomenon.

Chapter 8

Interlude: Can We All Just Get Along?

The learning theories reviewed in the preceding chapters come from different research traditions, which use different kinds of research methods and base themselves on different ontological and epistemological assumptions. The final sections of Chapters 5, 6, and 7 show that the traditions and theories are in conflict with each other, with some heavy criticism flying in all directions. It makes sense to pause for a moment to consider whether it is sensible, or indeed possible, to make use of all these frameworks simultaneously.

Section 8.1 below briefly compares the theories and traditions that I introduced in the previous chapters. In Section 8.2, I endorse combining theories to inform educational research from several perspectives.

8.1 There are commonalities and tensions between the learning theories

Figures 8.1 through 8.4 list some claims about learning originating from cognitive psychology[1] (Chapters 4 and 5), constructivism (Chapter 6), and phenomenography (Chapter 7). The reader should note that each of the three headings within these tables corresponds to a broad area, and that these broad areas overlap with each other – there are many proponents of cognitive constructivism, for instance. My intention is not to make sweeping statements and claim, for example, that all constructivists believe every one of the claims I have attributed to constructivism. I merely seek to highlight a selection of significant claims that originate from, and have relatively wide support within, each tradition.

The three traditions are in agreement on some points, in disagreement on others. Sometimes their claims are orthogonal with each other.

Ontological and epistemological issues (Figure 8.1) are at the heart of many of the arguments between proponents of the various theories and research traditions; these concern the nature of learning as well as the trustworthiness of scientific knowledge and the research methods suitable for rigorous research.

When it comes to how learning takes place, each tradition has its own points of emphasis (Figure 8.2). The cognitivist theories that I have reviewed tend to be the most detailed and specific in this respect. Phenomenographers tend to focus on the preconditions for learning which are realized in the variation present in the learning situation (as opposed to the generic processes that take place when the learner learns something). Many constructivist writers are very vague about knowledge representation and the processes of knowledge construction; some consider these to be idiosyncratic. Some constructivists – often with one foot in the cognitivist camp – make use of schema theory or conceptual change theory (Section 6.5).

On the pedagogical side, there are clear tensions between the three camps, especially between strong forms of constructivism and others (Figure 8.3). Where constructivists tend to encourage authentic, complex learning activities that are carried out in collaboration with others and chosen by learners themselves to correspond to their own goals, various cognitive psychologists (Section 6.8) are careful to remind us of the importance of 'managing' learning so that cognitive load does not become excessive and is fruitfully employed through practice to generalize from instances to increasingly automated schemas. This does not always need to involve an authentic, complex context.

[1] In Figures 8.1 through 8.4, "cognitive psychology" refers to schema theory, mental model theory, and cognitive load theory in particular.

Figure 8.1: Views from three traditions: the foundations of learning and research

Cognitive psychology
Knowledge about the world is represented mentally.
The mind is (metaphorically) an information-processing machine.
Working memory is very limited, which constrains simultaneous manipulation of knowledge.
Reliable scientific knowledge is gained primarily through hypothesis testing, experimental setups, and quantitative analysis.

Constructivism(s)
Knowledge is a subjective (or social) construction.
Different individuals/societies have different knowledge and different 'truths'.
Knowledge may not reflect any external world.
Qualitative, interpretive inquiry is (also) needed to gain scientific insight.

Phenomenography
Knowledge resides in experience, which is an inextricable two-way relationship between person and world.
Any phenomenon is understood only in a small number of qualitatively different ways.
Humans have a very limited capability for simultaneous focal awareness of phenomena and their aspects.
Qualitative, interpretive inquiry can illuminate human experience on a collective level.

Figure 8.2: Views from three traditions: mechanisms and processes of learning

Cognitive psychology
Excessive cognitive load on working memory prevents learning.
Domain-specific schemas (mental representations of concepts and patterns) are a key ingredient of expertise.
Abstraction to schemas facilitates chunking: ever larger elements can be dealt with as a single chunk.
Lengthy practice leads to ever more automated schemas which no longer strain working memory.
People are guided by their mental models of particular objects or systems; these models are often simplistic and incorrect.
To successfully simulate a system's behavior, a robust mental model of the system is needed.
With increasing experience, initial mental models of specific systems generalize to abstract, transferable schemas.

Constructivism(s)
Specifics of knowledge construction are often left vague; however, new knowledge is seen as crucially building on prior knowledge.
Some constructivists make use of cognitive theories such as schema theory. Some advocate conceptual change theories.
Some emphasize the plurality of perspectives to learning.
Social constructivists emphasize the interpersonal nature of learning and its situatedness within the authentic practice of communities.

Phenomenography
Each phenomenon has certain educationally critical aspects; each way of experiencing a phenomenon can be described in terms of these aspects.
Each such critical aspect is characterized by a dimension of variation.
Discerning variation along a dimension leads to discerning the critical aspect and to new ways of experiencing the phenomenon.
The most significant form of learning is learning to experience phenomena in new, more powerful ways that permit more powerful ways of acting.
Learning proceeds from understanding vague wholes to understanding ever more detailed parts within the wholes.

Figure 8.3: Views from three traditions: pedagogy

Cognitive psychology
Pedagogy should involve practice that helps the formation and automation of schemas.
Decontextualized, deliberate practice is a valuable tool in teaching.
Emphasizing patterns and similarities between cases can aid in schema formation.
Cognitive load should be managed through the careful sequencing of examples and problems.
Complex, authentic problems often lead to cognitive overload.
Active engagement with learning content helps learners form better mental models.
Teachers should seek to find out students' misconceptions about important content and to correct them.

Constructivism(s)
Learners should have a driving role in goal-setting; teacher- or institution-given norms for education are questionable.
Students should be heavily involved in the design of learning activities and their assessment.
Learning works best in rich and complex problem-driven contexts.
Learning tasks should reflect authentic practice by experts.
Learning should involve collaboration between students and their peers, and between students and experts.
Misconceptions or alternative frameworks are inevitable and can be used as a basis for further learning.
Teachers should explore students' prior knowledge and leverage it in teaching.

Phenomenography
Some ways of experiencing phenomena are better than others; education should have norms against which understandings can be judged.
Suitable pedagogies must be discovered separately for particular disciplines and particular subject matter.
A learning situation provides a space of variation in which certain variation can be discerned but other variation cannot.
The teacher's task is to find the best way to make it possible and likely for students to experience the critical variation.
Learning situations should engage the students' interest to explore the variation that is present.
Teachers should explore students' understandings of specific phenomena to discover the critical dimensions of variation.

Figure 8.4: Views from three traditions: programming education

Cognitive psychology
Both novices and experts may use top-down and bottom-up strategies for program comprehension, depending on schema availability.
In the absence of high-level schemas, novices have to work at a lower level more often than experts, and need to mentally trace programs at a lower level of abstraction.
Students have many misunderstandings of very fundamental programming concepts, such as variables and references.
Many of the problems students have involve the dynamics of program execution and the role of the computer (a notional machine) in the process.
Many novices do not know how to mentally trace program execution, nor are they much inclined to do so.
A robust mental model of a notional machine is important for many debugging tasks.
A conceptual model of a notional machine is useful as a teaching aid.

Constructivism(s)
Programming education should involve active work on complex projects that are as realistic as possible.
Novices often have no effective prior knowledge of how the computer works.
Students construct idiosyncratic knowledge of how the computer works at lower levels of abstraction than the code level.
Teaching in CS1 should address a lower level of abstraction than the code level the learners normally operate at.
Program dynamics in particular are a source of many misconceptions, which are troublesome because the computer is an accessible ontological reality that reacts to novice misconceptions in an unforgiving manner.

Phenomenography
Sometimes, novice programmers see programs merely as pieces of text, and programming merely as writing instructions.
One early challenge for the novice programmer is to go beyond this static perspective and to understand programming from other perspectives.
In particular, learning to program involves learning a new way of thinking in terms of what the computer does at execution time.
Other challenges include seeing the relationships between code and the real-world problems that programs solve.
Novices also understand various programming concepts (e.g., object, class, variable) in limited ways that restrict programming ability and further learning.

There is substantial agreement between the traditions, too. There is widespread agreement, for instance, on the usefulness of engaging learning activities that require the learner to take an active role in manipulating the content that is to be learned. Further agreement can be found where the theories have been applied to learning computer programming (Figure 8.4). Cognitivists, constructivists, and phenomenographers alike have reached the conclusion that a, if not the, crucial challenge for many novices at the CS1 level is to come to understand the relationship between program code and the computer, which is realized in the run-time dynamic behavior of programs.

Finally, I should mention that within each of the three traditions I have discussed, there exists a movement that emphasizes the social aspects of learning. This is perhaps most prominent within constructivism, where social constructivism and situated learning theory are highly influential. I admit the value of socio-cultural theories of learning as an additional perspective, but my focus in this thesis is not on the social aspects of learning. I will briefly discuss – mostly criticize – my work from the perspective of situated learning theory in Section 14.5.

8.2 Multiple theories give complementary perspectives

The ontological and epistemological assumptions of some schools of thought are difficult or impossible to reconcile, except by accepting them as alternative frameworks for thought, which can be compared and contrasted to gain new insights. The mental representations of a cognitive psychologist and the monist mind–world relationships of a phenomenographer, for instance, are two explicitly different ways of looking at the nature of knowledge, each with its own strengths and issues. Nevertheless, each of these views is fundamental to a research tradition that has expanded our understanding of what it takes to learn in general, and what it takes to learn computer programming in particular.[2]

The strengths of schema theory, cognitive load theory, and much of educational psychology lie in their detailed and rigorous consideration of the general processes and mechanisms of thought and learning. The strengths of phenomenography lie in the careful exploration of the ways in which particular contents of learning are crucial to the success or failure of the educational enterprise. As for the various flavors of constructivist theory, their main contribution to present-day education is arguably as torch-bearers for certain pedagogies which have previously been underappreciated, such as active and collaborative learning and learner empowerment.

Where findings from different research traditions point in the same direction, they strengthen each other. Where they point in different directions, they give us food for thought and remind us that learning is complex and multilayered. Often, different theoretical perspectives complement rather than conflict with each other, and something may be learned from points of conflict, too.

Multiple learning theories can also suggest, for projects such as my present one, different kinds of research approaches that complement each other; Thota et al. (2012) argue that drawing from different paradigms is useful, even necessary, for the present-day computing education researcher. According to the pragmatist Charles Sanders Peirce, "reasoning should not form a chain which is no stronger than its weakest link, but a cable whose fibers may be ever so slender, provided they are sufficiently numerous and intimately connected" (Menand, 1997, pp. 5–6, from the 1868 original). In this spirit, I take an eclectic and pluralistic view of learning theory. This view is consistent with the pragmatic perspective on mixed-methods research advocated by Johnson and Onwuegbuzie (2004), who draw on classical pragmatists such as Peirce. I am inspired by multiple learning-theoretical perspectives, and will make use of multiple research approaches in my own empirical work in Part V of this thesis.[3]

End of interlude. Now, one more theoretical framework of learning.

[2] There are of course still other perspectives on learning, beyond what I can cover here, such as those provided by biochemistry and activity theory, for instance (see, e.g., Jonassen, 2009). The traditions I have covered in my review are arguably the ones that have had the greatest influence on research on introductory programming education.

[3] Am I sitting on fences? I prefer to think of it as a small contribution to fence-lowering.

Chapter 9

Certain Concepts Represent Thresholds to Further Learning

An introductory programming course has a lot of content to cover in a limited time. What makes matters worse is that often a student gets stuck. There is something in the curriculum that he just cannot seem to get past. He cannot cope with the content that follows because he is 'thinking about it the wrong way'; he feels frustrated, insecure, and angry. Too often, the teacher forges ahead to new topics, leaving the student trailing.

Meyer and Land (2003, 2006) propose that embedded within academic disciplines there are troublesome barriers to student understanding, which they term *threshold concepts*. These "jewels in the curriculum" represent transformative points in students' learning experiences that allow them to view other concepts in a different light. Proposed threshold concepts in programming include program dynamics, information hiding, and object interaction.

Threshold concepts (TCs) are still a fairly young theoretical framework that requires better empirical support. However, they are obviously a fruitful basis for discussion and pedagogical explorations, as evidenced by the rapid growth of TC literature over the past few years. The ongoing work on threshold concepts may help us gain further insights into why some students seem not to learn 'any of the stuff', while other students seem to get 'all of the stuff' – a phenomenon familiar from CS1 courses. The answer may lie in the curriculum itself: TC theory suggests that some particularly transformative and integrative 'stuff' leaves many students stuck and unable to proceed until they are able to see the connections between related concepts.

Section 9.1 is a brief overview of threshold concept theory. In Section 9.2, I comment on the challenges of identifying threshold concepts. Section 9.3 reviews the work within CER that has sought to identify programming-related threshold concepts. Finally, Section 9.4 considers some pedagogical implications.[1]

9.1 Mastering a threshold concept irreversibly transforms the learner's view of other content

A threshold concept is not a mere 'core concept'. To qualify as a TC, a concept must meet a stricter definition, albeit one which is still being debated. Meyer and Land (2006) list five characteristics that a threshold concept is likely to have.

- A threshold concept significantly *transforms* how the student perceives a subject or part thereof, and perhaps even occasions a shift of personal identity;
- it is probably *irreversible* so that the transformation is unlikely to be forgotten or undone;
- it *integrates* other content by exposing its interrelatedness;

[1]This chapter has been adapted from an earlier publication. Large swaths of text have been reproduced from: Juha Sorva: "Reflections on Threshold Concepts in Computer Programming and Beyond", in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pp. 21–30 http://doi.acm.org/10.1145/1930464.1930467

- it may *mark boundaries* in 'conceptual space' between disciplines or schools of thought; and
- it is potentially very *troublesome* to students for any of a variety of reasons including conceptual complexity, tacitness in expert practice, apparent meaninglessness, and counter-intuitivity.

The literature attributes some additional characteristics to threshold concepts. I will briefly discuss three: 1) crossing a threshold involves a state of liminality; 2) the resulting transformations lead to new ways of thinking and practicing, and 3) threshold concepts are often associated with generic 'everyday' ideas.

Liminality

A *liminal space* is the transitional period between beginning to learn a threshold concept and fully mastering it (McCartney et al., 2009; Meyer and Land, 2006). While some learners cross some thresholds quickly (as a single a-ha! moment) and sometimes effortlessly, liminality often lasts for a considerable length of time. The liminal space is a fluctuating place of transformation: students in liminal spaces tend to oscillate between old and new states and experience strong, often negative emotions, and may attempt to mimic the behavior of others whom they perceive as having crossed the threshold already. Meyer and Land argue that *pre-liminal variation* – variation in students' perceptions of a threshold concept as the concept first 'comes into view' – plays a key role in how and why some students negotiate liminal spaces productively while others struggle and may give up altogether.

Ways of thinking and practicing

A *way of thinking and practicing* within a discipline or subject area (or community of practice; cf. Section 6.6 above) involves concepts, forms of discourse, values or indeed "anything that students learn which helps them to develop a sense of what it might mean to be part of a particular disciplinary community" (McCune and Hounsell, 2005). Many authors have commented on the apparent relationship between threshold concepts and ways of thinking and practicing: mastering a TC provides a new 'lens' through which to view the subject, opening up new avenues for thought and action (Meyer and Land, 2006; Land and Meyer, 2008; Moström et al., 2009; Zander et al., 2009). Conversely, failing to internalize a TC and make it a part of how one thinks and acts leaves the student stuck. Without crossing the threshold, the student cannot perceive other concepts, new problems, potential solutions, and indeed the very subject they are studying, like a full-fledged member of the disciplinary community would.

Transforming ideas from everyday life

Another framework for curricular analysis that may be used alongside threshold concepts was formulated by Schwill (1994) from the earlier work of Bruner (1960). *Fundamental ideas* are lasting, general ideas of broad, perhaps universal significance. They connect topics within and beyond the borders of an academic discipline. The distinguishing feature of a fundamental idea is its wide applicability – across disciplines, across levels of education, across time, and across the divide between academic pursuits and everyday life. Algorithmization, language, abstraction, and state, among others, have been proposed as fundamental ideas of computer science – they characterize the discipline and yet are applicable beyond its confines (Schwill, 1994; Sorva, 2010).

Fundamental ideas have an 'everyday' character. Building from an observation by Zander et al. (2008), I have suggested that threshold concepts often involve the transformation of one or more fundamental ideas into discipline-specific forms (Sorva, 2010). For instance, the (possible) threshold concept of information hiding is a form of the universal notion of abstraction that is central to how computer scientists think (cf. Colburn and Shute, 2007). The everydayness of the fundamental idea(s) involved in a TC means that students have intuitive pre-liminal understandings of aspects of a TC. Everyday understandings are often 'obvious' and tacit, and may be difficult to change. Shinners-Kennedy (2008) provides a good discussion of how the notion of state – quite common and unproblematic in everyday life – becomes troublesome once the student has to make it central to their conscious thinking and relate it to programming.

9.2 Identifying threshold concepts is not trivial

Meyer and Land (2006) use very cautious language as they describe the characteristics of threshold concepts. A TC "is likely" (p. 7) to have the five main features listed in the previous section; it opens up a way of thinking that "may represent" (p. 3) how people think within a particular discipline or about particular phenomena; it is "often more" (p. 101) than just a core concept, and so forth. Additionally, Meyer and Land do not clearly define what they mean by "concept".

Meyer and Land's caution, although it has been criticized (Rowbottom, 2007), is understandable, since the threshold concept theory is still a young one. Moreover, constructivist learning theory (Chapter 6) suggests that people have different kinds of knowledge structures, and what is a threshold for someone may not be one for someone else – a notion that has also been pointed to as a weakness of the threshold concepts framework (Rowbottom, 2007). However, as nature restricts our knowledge-constructing activities (Section 6.3), certain concepts may be particularly troublesome and transformative (etc.) to most people.

Thought-provoking though it is, Meyer and Land's fairly loose definition of threshold concepts does not make the job of identifying TCs very easy. A variety of interpretations exists concerning what counts as a TC. As I see it, there are two main issues: the use of the word "concept", and different opinions as to how well a candidate concept has to fulfill each of Meyer and Land's five main criteria.

A threshold what?

Various scholars operating within the threshold concepts framework have found it useful to consider educational thresholds that are not necessarily individual concepts with commonly accepted names, but something broader. Lucas and Mladenovic (2006) suggest that learning accounting involves a "threshold conception" – a transformation of world-view not associated with a single concept. Savin-Baden (2006) argues that problem-based learning may become a "threshold philosophy". Some authors simply avoid the word "concept", and speak merely of thresholds. One recent trend in threshold concepts research, suggested by Davies and Mangan (2008), is to characterize disciplinary ways of thinking not so much via individual concepts, but as "webs" of interrelated TCs.

A related issue is that one does not simply 'get' or 'not get' a concept – one gets it in a particular way (cf. Chapter 7). Crossing a threshold means understanding something – which may previously have been understood differently – in a way that opens up a powerful new perspective.

My use of the word "concept" in "threshold concept" may be read as shorthand for "way of understanding certain curricular content".

How TC does it need to be?

Clearly, the concept of pointer does not integrate as much content as the concept of information hiding. Does that make it less of a threshold concept? How widely does a threshold concept have to integrate? How radical does the resulting transformation have to be? Should we keep the bar high, with fewer TCs per subject area, or low, with a smaller difference between a core concept and a TC? Some suggested threshold concepts (in computing and in other disciplines) are claimed to be responsible for large-scale transformations of students' views of an entire discipline, while other 'local thresholds' only integrate a handful of neighboring concepts. Rountree and Rountree (2009) have noted how difficult it is to agree on the granularity of threshold concepts, and to expose possible hierarchies of threshold concepts at different levels of granularity.

For me, much of the promise of the threshold concepts framework lies in how a threshold concept is curricular content that is not just another bit of content, but provides a way of dealing with a lot of other concepts. Of greatest interest are those concepts whose effect is the greatest. As Perkins (2006) puts it, TCs include concepts that are more than particularly tough conceptual nuts – some threshold concepts shape students' sense of entire disciplines or large subject areas, giving them access to new ways of reasoning, gaining knowledge and problem solving. Below, I set the bar high, as I discuss candidates for 'big' TCs that change how students think about and practice computer programming, and that provide them with new tools for tackling many kinds of programming problems.

Telling TCs apart from other content

I have argued (Sorva, 2010) that if we are to learn about the nature of threshold concepts and identify them, we need to learn to discern contrasts between TCs and other forms of curricular content. One such form is fundamental ideas, introduced above. The gently developing, extremely broad nature of fundamental ideas contrasts with the drastically transformative, troublesome, irreversible, discipline-specific nature of TCs. Identifying fundamental ideas may also help the search for threshold concepts in that transformative thresholds may be found where fundamental ideas from everyday thought 'meet' a discipline.

Another non-TC form of content is what I have called a *transliminal concept* (Sorva, 2010), that is, a concept that 'lies across the threshold'. A transliminal concept is not in itself transformative or crucial for developing a new way of thinking and practicing, although it may certainly be a core concept which is well worth understanding. However, mastering a transliminal concept is predicated on first acquiring a new way of thinking and practicing by crossing the threshold. Learners may struggle with transliminal concepts, sometimes not because the concept itself is so troublesome, but because their progress is blocked by the barrier of the prerequisite threshold concept. Many transliminal concepts can be associated with a single threshold concept; this is indeed likely to be the case, as a threshold concept has wide applicability within a discipline.

9.3 The search is on for threshold concepts in computing

Learning to program is not just the accumulation of commands, templates, and strategies, but the development of a new way of thinking (Section 7.5). In the words of one of the students quoted by Moström et al. (2009), one needs to become more than "just someone typing in code". The threshold concepts framework suggests that coming to grips with certain content is decisive in transforming a 'code typist' into a programmer. But which content? In Section 9.3.1 below, I argue that program dynamics represents a 'big' transformative threshold that beginners must cross. Section 9.3.2 reviews other candidate threshold concepts within programming.

9.3.1 Program dynamics is a strong candidate for an introductory programming threshold

A crucial distinction in programming is the one between the existence of a program as code and its existence as a dynamic execution-time entity within a computer. Code is tangible and its existence is easy to perceive. The existence of the latter aspect of a program, which I will call *program dynamics*, is much less so.

Integration

A dynamic view of a program brings together program code, the state of the program, and the process that changes it, as well as the computer on which the program runs (if not the actual hardware, at least a notional machine; see Section 5.4). The ability to view programs as dynamic is required to genuinely understand a legion of transliminal concepts and distinctions: variables and values; function declarations vs. function calls; classes, objects, and instantiation; expressions and evaluation; static type declarations vs. execution-time types; scope vs. lifetime; etc. The dynamic use of memory to keep track of program state is central to much of this integrative power (cf. Vagianou, 2006; Shinners-Kennedy, 2008).

Transformation

Learning to see a program in dynamic terms transforms a learner's understanding of programming concepts. A phenomenographic study by Eckerdal and Thuné (2005) showed that there is an educationally significant qualitative difference between seeing an object as a piece of code and seeing it as something active (dynamic) in a program. Even more pertinently, the same authors found a more generic difference between how students see computer programming as writing code or as a way of thinking that relates program code to what happens during execution (Thuné and Eckerdal, 2009, and see Section 7.5).

Program dynamics further takes the everyday notion – fundamental idea – of state and transforms it into something central in how the programmer thinks. A dynamic view of programs leads to what Perkins (2006) calls a new *episteme*, a new way of reasoning about programs that is impossible unless the student has ingrained the notion of program dynamics into their thoughts and practices. In particular, thinking in terms of program dynamics makes possible the key skill of program tracing: stepwise reasoning about runtime behavior in terms of what the notional machine does as it executes a program (Section 5.5).

Trouble

Program dynamics are troublesome. Practically any student of programming can pay lip service to the idea that programs are executed step by step, making things happen within the computer. However, not all of those students genuinely internalize this notion and make it work for them. We have seen in the preceding chapters plentiful evidence of novices failing to learn what happens when a program is run. From a threshold concepts perspective, Vagianou (2006) comments on the use of memory in programming, noting that novice programmers may not "realize *how* such use takes place and their active role (through the program) in this process". In light of the evidence, it is not too surprising that novice programmers often do not systematically trace the execution of their programs, sometimes because they do not know how, sometimes because they fail to perceive that as a useful pursuit (Section 5.5).

Part of the troublesomeness of program dynamics lies in its tacitness. As I noted in Section 5.5, programmers rarely make explicit the dual nature of programs, which is obvious to them – when we speak of a "program" we refer to either the code, or to what the code does upon execution, or to both at once. The centrality of the previously unproblematic notion of state to program dynamics may also be counter-intuitive to novices (Shinners-Kennedy, 2008).

Boundaries

Vagianou (2006) points to how end users and programmers have different stances towards computer programs. Only the latter group has a sense of being directly involved in what happens when a program gets executed by a computer. This is one way in which program dynamics serves as a boundary marker, separating computer programmers from non-programmers. A student who does not trace programs or think about the dynamics of their execution is not really thinking and practicing like a computer programmer.

Program dynamics also demarcates two schools of thought *within* computing: it lies at the border of computer programming and programming-as-mathematics. The latter episteme, which Dijkstra famously and controversially advocated as the perspective of choice for CS1 courses, is ruled by formal logic and proofs, and the former by testing, mental tracing, and operational reasoning (Dijkstra vs. al., 1989, and see Section 14.5.3).

Irreversibility

I have little to offer in the way of research-based evidence of irreversibility. However, I have never heard of a programmer forgetting how to see programs as dynamic, traceable entities once they have made that concept their own, nor do I expect to hear of one. A sign of irreversibility may also be the difficulty that some experienced programmers have in perceiving how programming appears to the novice who has not yet crossed this early 'obvious' threshold.

9.3.2 Other programming thresholds have also been suggested

Much of the work on threshold concepts within CER has been done by various lineups drawn from the multinational "Sweden Group" of researchers. In an early paper, Eckerdal et al. (2006a) discussed the relationship of threshold concepts to other theoretical frameworks that are influential within the context of computing. Based on the literature, they suggested abstraction and object-orientation as two possible threshold computers in computing. In their later work, they found evidence in student interviews to support the claim that object-oriented programming and pointers could be threshold concepts (Boustedt

et al., 2007). The latter was later rephrased as "memory/pointers" (Zander et al., 2008); I have suggested (Sorva, 2010) that the pointer may be a transliminal concept for the TC of addressable memory.

Moström et al. (2008, see also Thomas et al., 2010) returned to the candidate TC of abstraction and argued on the basis of their data that abstraction *in general* does not appear to be a threshold concept even though they found evidence of transformative, integrative and troublesome experiences that students have had with specific forms of abstraction in software design and implementation. Building on their work, I suggested information hiding as a threshold concept that transforms the fundamental idea of abstraction to a discipline-specific form (Sorva, 2010).

The "Sweden Group" have also found evidence supporting the existence of liminal spaces in computing education (Eckerdal et al., 2007; McCartney et al., 2009) and described the transformations of identity and ways of thinking and practicing that take place as computing students learn threshold concepts (Moström et al., 2009; Zander et al., 2009).

Vagianou (2006) proposed program–memory interaction as a threshold concept in introductory programming; my discussion of program dynamics extends her argument. Reynolds and Goda (2007) suggested that the pervasive themes given in ACM curricula (e.g., abstraction and professionalism) fulfill the criteria for threshold concepts. As discussed above, such topics might be better interpreted as fundamental ideas; the same goes for Shinners-Kennedy's (2008) suggestion of state. Flanagan and Smith (2008) contended that in introductory programming, the nature of programming languages is an overall threshold that students need to overcome before being able to tackle smaller "local thresholds" (transliminal concepts?) such as the Java interface construct.

Rountree and Rountree (2009) critically review the literature on threshold concepts and conclude by emphasizing how expert computer scientists, rather than students, should be focal in the ongoing effort to identify threshold concepts. They mention generics and recursion as possible TCs within computing. Holloway et al. (2010) considered recursion as well as object-orientation to be threshold concepts as they sought to develop a quantitative instrument for assessing whether a concept is a TC or not. I have suggested that the idea of object interaction – objects collaborating through message passing – is a TC that opens up to the object-oriented paradigm (Sorva, 2010; see also Sien and Chong, 2011).

9.4 Pedagogy should center around threshold concepts

The programming curriculum is a conceptual space inhabited by entities of various kinds. Teachers need to identify which areas of knowledge constitute transformative thresholds, which threads enter the curriculum as fundamental ideas, and what roles other important concepts may play in students' attempts to navigate the conceptual space. Different pedagogical solutions may be required for teaching TCs than for other content.[2] Teachers may influence the paths that students take, affecting when a TC is first seen, the angle from which it is initially viewed, and the transliminal concepts that are seen behind it. It is not likely that any sequencing of content is universally better than all the others, but the teacher who bears in mind the characteristics of different kinds of conceptual content is better positioned to make good decisions.

I will use the proposed TC of program dynamics as a running example as I consider some pedagogical implications of threshold concepts.

9.4.1 Threshold concepts demand a focus, even at the expense of other content

From a pedagogical point of view, threshold concepts are supremely important, as lack of mastery of a threshold concept renders further teaching inefficient if not entirely fruitless. In Cousin's (2006) words, the threshold concepts framework suggests "a less is more approach" to teaching: the teacher should spend time and effort on the selected concepts that transform the student's view of the discipline and make learning other concepts easier, rather than burying these conceptual jewels within a vast bulk of knowledge where they may go all but unnoticed.

Recursion, reference parameters, and instantiation are concepts that are hard enough to grasp even for a novice who is capable of thinking about programs in terms of their dynamic aspect and tracing

[2]For instance, it might be that fundamental ideas are initially best approached by fostering learners' everyday intuitions (as suggested by Bruner, 1960), but that TCs require actively challenging established everyday thinking in pedagogy.

execution step by step. Without the TC of program dynamics, understanding those other important concepts becomes next to impossible. A programming teacher must not fall into the trap of assuming that students think as the teacher does. Instead, teachers need to help students uncover the nature of the tacit "underlying game" (Meyer and Land, 2006).

Perkins (2008) notes that there is a cost to be paid for teaching TCs – just as they are hard to learn, they are hard to teach – but that it is a cost that is generally worth paying. TCs demand an emphasis in teaching and assessment, even at the expense of other concepts. A student who passes a programming course without having developed a dynamic perspective on program execution has not really learned very much about computer programming, no matter how many concept definitions they may have memorized or how many code templates they may be capable of applying. Conversely, someone who has crossed the threshold is well positioned to learn more with relative ease.

Neither the teacher nor the student should be allowed to settle for mere lip service to the idea that programs run step by step and use memory. Each threshold concept within a curriculum should be worked on enough to make sure that students genuinely work them into their ways of thinking and practicing. This requires time: the student who is rushed may experience "psychological vertigo" (to borrow an expression from Booth, 2006), trip on the threshold, and fall flat on their face.

9.4.2 Teachers should make tacit thresholds explicit and manipulable

The threshold concepts literature makes many pedagogical suggestions (Meyer and Land, 2006; Land and Meyer, 2008). For instance, teachers should seek to inform students about the existence of threshold concepts and liminality, increase students' metacognition about their liminal states, and help them deal with uncertainty and the emotional issues involved. Students should be helped to become aware of the ways in which they presently think and practice, and motivated to transform those ways. The kinds of mimicry that are motivated by a genuine attempt to cross a threshold should be seen as positive rather than negative. Students need to be engaged in actively and consciously manipulating each threshold concept in order to internalize it.

Consider again the example of program dynamics. CS1 teachers should try to help their students become aware of the importance of this concept, its possible troublesomeness, and how the students themselves think about programs and reason about what programs do. One way to accomplish this may be through tools, visualizations, and metaphors that concretize the dynamic aspects of programs. By making program dynamics visible, the teacher may not only help the student to think about programs dynamically but make the student more aware of what they are doing. A student who is aware of thinking about program execution as a dynamic step-by-step process – perhaps in terms of a visualization – may find it easier to grasp the general principles embodied in the TC and relate them to multiple contexts.

Meyer and Land (2006) call for a pedagogy of threshold concepts that engages the student in manipulating the conceptual material. This can be a challenging task, since threshold concepts tend to be abstract and tacit. Program dynamics is no exception: as noted, many students appear to be not only unable but disinclined to trace the dynamics of their programs. Students should be placed in situations where they feel they *need* a new way of thinking – a threshold concept – in order to explain something they wonder about or to solve a pertinent problem. Marton and Booth (1997) speak of building a *relevance structure* for a learning situation – defined as a person's experience of what a situation calls for – and encourage teachers to "stage situations for learning in which students meet new abstractions, principles, theories, and explanations through events that create a state of suspense". There are many ways to build a relevance structure for program dynamics, for instance. Having students confront the reasons for bugs in their own programs is one option; another is requiring students to predict program behavior and face the fact that their predictions often fail. To build relevance structures, teachers may employ specific transliminal concepts such as pass-by-reference and instantiation. These concepts puzzle students and cannot be fully understood without a shift of perspective. Transliminal concepts may serve as 'educational macguffins' that draw students into and through the 'main plot' of learning a threshold concept. For students to eventually appreciate the power of the generic TC, a combination of multiple transliminal concepts may be needed.

Part III will present many specific examples of making program dynamics visible.

Part III

Teaching Introductory Programming

Introduction to Part III

The learning theories reviewed in Part II have put forward many recommendations on teaching, most of them in fairly generic terms. How do those recommendations translate into something more tangible? What concrete practices and recommendations for teaching CS1 have been made – whether inspired by some learning theory or teachers' personal experience and intuition? What different approaches to teaching CS1 are being used?

The two chapters in Part III examine these questions, the first with a broader scope and the second with a narrower one. In Chapter 10, I take a look at several prominent themes in scholarly debates on CS1 teaching approaches. One of the topics of Chapter 10 is the role of the notional machine; we will see that those teachers who explicitly teach about the execution-time dynamics of programs often make use of visualization. Software tools for visualizing notional machines are the topic of Chapter 11. I review existing systems and their empirical evaluations, and discuss the possibly key role of user interaction in the success of software visualization as a learning aid.

Chapter 10
CS1 is Taught in Many Ways

The literature on teaching CS1 is large. Popular topics in practice papers and research papers on CS1 include language and paradigm choices, different kinds of assignments and assessments, the ordering of topics in the curriculum, technology-supported learning, pair programming, gender issues, motivational techniques, predictors of student success, student-driven pedagogies, and more. In this chapter, I focus on a few topics that give us a big picture of CS1 education or are otherwise relevant for the upcoming discussion of visual program simulation in Part IV.

Section 10.1 tries to convey a sense of what kinds of approaches to teaching CS1 there are, in broad terms. I especially focus on the arguments for complex, authentic, limited-guidance problem solving in CS1 versus carefully managed instructional design with less learner control. Section 10.2 is a very brief review of approaches that seek to enhance CS1 by exposing expert programmers' schemas and processes. Section 10.3 discusses how (and if) teachers have approached the question of teaching about a notional machine; visualization is identified as a relatively popular approach. Finally, Section 10.4 reviews the hottest debate in CS1 education in recent years – the role of object-orientation – and relates it to our discussion of notional machines.

10.1 Keep it simple, or keep it real?

Table 10.1 lists features of CS1 pedagogies, paired up as opposites.

Generally speaking, the strategies on the left are more 'student-centered' in the sense that learning goals and activities are chosen and managed by students rather than teachers, and more complex, in the sense that the learning tasks are larger, more complicated, and more realistic. Many of these strategies match constructivist recommendations (Chapter 6).

In the strategies on the right-hand side of Table 10.1, the teacher tends to exert more direct control over the goals and activities; they might be termed 'teacher-centered' but also 'learning-centered', as the goal is to ease the learning process and maximize students' achievement of learning goals. Sometimes, the use of these strategies is motivated by the recommendations for instructional design derived from educational psychology.

Table 10.1 does not mean that there are two main ways to teach CS1, and certainly is not meant to imply that all the items on one side go with each other. Any given CS1 offering is almost certain to feature a mix of strategies from each side of the table and compromises between the polar opposites that I have listed.

The subsections below consider in more detail some of the themes present in Table 10.1.

10.1.1 Complexity is good

Just as CS1 courses traditionally have goals that are relatively challenging from a cognitive point of view (see Chapter 2), they also commonly center around a relatively complex learning activity, namely, solving problems by writing programs.

The traditional way to teach programming is by having the students program – an intuitively appealing practice which is in line with advice from the literature of education as it aligns the goal with the teaching (Biggs and Tang, 2007). Practically any CS1 course is going to involve some coding, usually quite a lot of

Table 10.1: Some strategies for teaching CS1

Students practice real professional skills in authentic contexts.	Students start with fundamentals, eventually proceeding to real professional skills.
Students design programs.	Students examine or use given program designs.
Students write program code.	Students read program code.
Large and complex programming assignments.	Relatively small programming assignments.
Ill-structured programming projects; students must manage requirements and engage in simplifying complex problems.	Well-structured programming projects; students work to a specification given by the teacher.
Open-ended projects: tasks are expansible according to students' desires.	Closed projects: task scope is restricted by the assignment.
Groupwork, social participation, negotiation of goals and/or content.	Assignments solved individually.
New concepts are motivated through inquiry, e.g., "How is it possible to create a program that does X?"	Students are taught about the ways in which present content will be useful in the future.
Students select their learning goals.	The teacher selects learning goals.
Students solve problems themselves.	Students study examples of problem solutions.
Computing is presented in a contextualized way that takes students' future professions (e.g., engineering) into account.	Computing concepts are presented in a general, decontextualized way that is intended to transfer to different contexts.
Assessment based on authentic programming work in an authentic context.	Exams and other artificial assessment settings.
The teacher leaves it to the learners to determine what content is relevant and useful.	The teacher guides learners to relevant content.
Limited guidance: the teacher interferes as little as possible in the learning process.	Direct guidance: the teacher strongly controls the learning process to maximize its efficiency.
"Jump in at the deep end (or as near to it as you possibly can)."	"Wade in and take careful steps towards the deep end."

it and right from the beginning. Some teachers advocate design-first approaches in which novices design software components from the get-go, further adding to the authenticity and complexity of the learning activity (e.g., Alphonce and Ventura, 2002; Moritz and Blank, 2005).

Many forms of constructivism (Chapter 6) suggest that learning is most successful when learners are actively engaged in complex, authentic activities. Given the above, we can say that the majority of existing CS1 courses are – in this limited sense – constructivist. The extent to which this is true varies, as teachers make varying use of open-ended assignments, ill-structured problems, inquiry-driven pedagogy, groupwork, and other constructivist practices.

No reproduction, please (we're constructivists)

Some teachers introduce programming through ill-structured problems that the learners need to make sense of for themselves. The pedagogy of Greening and his colleagues exemplifies this standpoint well. They call for less explicit linking between the assignments given and the programming techniques needed to solve them:

> *The production of an ill-structured problem is likely to add an element of reality to the lab, and allows the students to have their own Eureka!s about the underlying nature of the exercise. We have seen courses where each lab is a straightforward instance of "write code using the language feature covered last week"; this does not support effective problem-solving.* (Fekete and Greening, 1996, p. 298)

Elsewhere, Greening (1999) is critical of assignments in which each student (or group) produces their own clone or near-clone of a solution that was already previously known to the teacher and that is much the same as every other student's solution. An example is a program-writing assignment which has to satisfy a precise set of teacher-given low-level specifications. Greening claims that "many existing courses appear to have an obsession with knowledge reproduction that borders on the petty" (p. 59) and cites automatic assessment of programming assignments as a symptom of this unsatisfactory pedagogy as "the desired product outcome is typically so trivial and predictable that it makes sense to have students submit their work for automatic marking as a matter of convenience" (p. 54).

Greening argues that we must not postpone solving interesting programming problems until later in the curriculum. His constructivist position is to bring complexity into the learning environment, which means that we should allow the students themselves to engage in structuring the complex problems presented, rather than have them simplified for them.

Problem-based learning

Problem-based learning (Section 6.4) is a quintessentially constructivist pedagogy that is increasingly being applied to different educational subfields, including computing education. PBL embraces complexity and fuses together many of the practices from the left-hand side of Table 10.1.

There is limited support for the suitability of problem-based learning for CS1 courses. A group of teachers and researchers from the University of Sydney report experiences and survey results on the use of PBL in introductory-level computing courses over several years (Kay et al., 2000; Fekete and Greening, 1996). Their results are tentatively positive regarding the development of generic skills and elimination of plagiarism, in particular. Greening (1999) concludes that this trial has shown that PBL "works well with first-year students with a range of prior computing experience". Nuutila et al. (2005) discuss how PBL can be adapted for learning programming and report a low drop-out rate among students who took a problem-based CS1. Kinnunen and Malmi (2005) report mixed findings from a series of experiments with PBL in CS1, and discuss at some length the characteristics of different PBL groups, some of which did well while others did quite poorly.

Problem-based learning continues to inspire the development of some introductory programming courses (e.g., O'Kelly and Gibson, 2006; Ambrósio and Costa, 2010), but a detailed picture of what it takes to successfully make use of PBL in CS1 is yet to emerge.

10.1.2 Complexity is bad

> *In teaching writing* [. . .] *one starts with writing sentences, then paragraphs, then essays. In computer science, an error has been made by assuming that the student should start out by writing the equivalent of a whole literary form.* (Buck and Stucki, 2000)

Some scholars within CER have called for pedagogy that helps students take small, carefully managed steps rather than immersing them too soon in the complexity of program design and writing code. Such advice tends to emphasize code-reading before writing, the use of numerous small program examples for deliberate practice, and the completion of partial programs before writing entire programs from scratch.

Different teachers justify their pedagogies differently. Some see the skill of reading code as a prerequisite for writing so that it makes sense to develop a reading ability first, with learning to read programs perhaps a more realistic goal for CS1 courses than the notoriously difficult goal of program creation. Others argue that reading and completion tasks are simpler and that they do not cognitively overload students but that they nevertheless they can serve the goal of learning problem solving (see Section 4.5). This line of thinking goes against the conventional wisdom that one learns programming best by (only) writing programs.

Van Merriënboer and Krammer (1987) presented a case for an introductory programming curriculum in which students primarily read, modify, and extend programs. Their approach has since been established as an example of the 'completion effect' of cognitive load reduction, which states that beginners benefit from completion problems more than from full-blown problem solving (see Section 4.5.3 above). Inspired by the work of van Merriënboer and his colleagues, Chang et al. (2000) implemented an introductory programming course that revolves around template-supported completion problems; Garner (2002) similarly designed a software-supported pedagogy of completion problems for CS1. According to Shaffer et al. (2003), the completion effect is not yet sufficiently recognized in programming education, despite its empirical support.

Carbone et al. (2000) observed various "poor learning behaviors" in their students and set out to address these by revamping assignments. Among their main recommendations is replacing some of the coding by alternative activities, such as tracing code and answering questions about it.

Buck and Stucki (2000) argue that having novices design their own programs is a folly. They "feel strongly that students learn better when they are provided a context that constrains their thinking in a directed fashion". Buck and Stucki recommend a progression of assignments that matches Bloom's taxonomy (Section 2.1), which culminates in design and the evaluation of designs only at the highest levels. Initially, program-writing tasks should require students to merely implement given designs or complete teacher-authored programs. A similar approach was used by Caspersen and Bennedsen (2007; Caspersen, 2007), who "adopt an incremental approach to programming education in which novices [. . .] initially do very simple tasks and then gradually do more and more complex tasks, including design-in-the-small by adding new classes and methods to an already existing design." Caspersen and Bennedsen put cognitive load theory and worked-out examples to use throughout their theory-driven CS1 course design. Their approach further portions complexity through the 'consume-before-produce principle' – students use given classes, methods, interfaces etc., before they define their own (cf. Pattis, 1993; Howe et al., 2004) – and through systematic simple-to-complex sequencing of the object structures that students encounter. Yet another approach is that of Thompson (2010), who suggests a reading-first strategy for teaching object-oriented programming that is inspired by variation theory (Section 7.3.2) and the BRACElet studies on reading, tracing, and writing code (Section 3.2).

Lister (2001, 2011a) goes a step further: he argues that students should not be required to write any original code at any stage of CS1. Only when students have built up the ability to reason about code in a sufficiently sophisticated manner will they benefit from designing their own code. To get students to this point, teaching should emphasize learning tasks such as explaining what given code does, implementing given pseudocode, and making small modifications to given code. Lister and Leaney (2003) used Bloom's taxonomy to structure assignments in a CS1 course so that to get a higher grade, students were required completion of tasks of a higher level of cognitive complexity (e.g., to get an A, a student would have to create an original program from scratch), while cognitively less demanding tasks such as reading code were enough to get lower grades (see also Burgess, 2005).

10.1.3 Guidance mediates complexity

In Section 6.8, I reviewed the general dispute concerning constructivist vs. direct instructional methods. Echoes of this hot debate can be observed in the CER literature as well. An issue that is central to it is the amount of guidance given to students. Within computing education, too, scholars have been keen to distance themselves from minimal guidance pedagogy, emphasizing the need for guidance even when adopting constructivist pedagogies. For instance, Hamer et al. discuss the criticism by Kirschner et al. of minimal guidance pedagogy (see Section 6.8) as they apply a constructivist contributing student pedagogy (CSP) to computing education:

> [Kirschner et al., 2006] *is an important review and critique paper that could be interpreted as providing evidence that CSP approaches are not likely to represent an advance on learning. This is not the case. CSP is not 'minimal instruction'; on the contrary, CSP activities typically require substantial instructor guidance and support to ensure success. This guidance may not be in the traditional form of preparation of materials or the static delivery of lectures, but may rather include pointing students in the right direction towards resources (and making sure that sufficient resources are available), fostering the development of student trust in the creation of a safe environment for students to present their contributions, fair negotiation of assessment processes and criteria, development of software or IT tools to support the CSP activity, and individual student support in an unfamiliar learning environment. Kirschner's paper reminds us that CSP is not a 'cheap option'; time and resources formerly spent on traditional content delivery now need to be deployed elsewhere.* (Hamer et al., 2008, p. 197)

Proponents of constructivist CS1 commonly do stress the need to mediate task difficulty in some way even as they allow complexity into the classroom (see, e.g., Greening, 1998; Ellis et al., 1998). Scaffolding can take many forms, from teacher-given hints to given learning materials or tools, or program code. For instance, novices may be able to finish or modify a given complex programming project – an authentic task! – even if they are not capable of designing or coding it from scratch (see, e.g., Kölling and Barnes, 2008). The given parts of a program may either be intended for the students to read or may be ignored by the student, depending on learning goals (a glass-box scaffold or a black-box scaffold, respectively, as defined by Hmelo and Guzdial, 1996). Moritz and Blank (2005) used an intelligent tutoring system to scaffold their design-first CS1.

Kinnunen and Malmi's (2005) experimentation with different forms of PBL also highlighted the need for significant guidance in a constructivist CS1: when less tutor time was available (or when tutor meetings were not obligatory) some groups did very poorly, whereas more guidance led to more consistent performance.

Balance?

Learning programming involves a complicated weave of content, cognition and cognitive load, motivation, problem solving, practice, and participation. When successful, it changes ways of thinking about programming and the world. There are benefits to both authentic complexity and simple manageable steps. The two approaches can be combined by mixing different kinds of activities and the clever use of scaffolding. In the end, the teacher who strikes a balance between student freedom and learning management that is well suited to his particular teaching context and content will probably have the greatest success.

10.1.4 Large-class introductory courses suffer from practical restrictions

Many introductory programming courses around the world have large numbers of students, often in the hundreds. At Aalto University, for instance, our two main CS1 courses currently take in over 400 and over 700 students, respectively; in the past, we have had to deal with classes of over a thousand students.

Different institutions have very different teaching budgets, but too often the number of students is not reflected in the number of teachers. What this means in practice is that the possibilities for one-on-one teaching or even small-group work are often limited. Teachers of large classes struggle to provide enough

guidance and feedback to their students. The matter is not helped by the fact that programming is a highly skill-oriented subject that requires a lot of practice. This practice can take the form of many small assignments or fewer large ones, but in either case the strain on resources is considerable – merely marking thousands of programming assignments can be a daunting task, not to speak of giving useful formative feedback to individual students.[1]

Problem-based → expensive?

When the teacher-student ratio is low in a CS1 course, using methods such as problem-based learning is potentially problematic. Open-ended, complex, and authentic tasks require a great deal of teacher guidance to work, and guidance is expensive.

Shipman and Duch (2001) – who taught physical science, not computing – report that PBL can work in large classes but warn that their teaching worked significantly better in a "large" class (of 120 students) than in a "very large" class (of 240). Within computing education, Kay et al. (2000) used PBL with apparent success in a very large course, and in fact cite problems with large classes as their motivation for adopting a problem-based approach. However, what little other evidence there is of the applicability of PBL to CS1 suggests that the availability of significant amounts of guidance from experienced tutors is key to success (Kinnunen and Malmi, 2005), which means that a sizable teaching budget is needed to implement PBL for hundreds of students. Carbone and Sheard (2003) have reported promising results with a 'studio approach' to CS1 that resembles PBL in that it emphasizes groupwork and integrates programming into other curricular content; however, they acknowledged that the approach adds to the expense of teaching. Bareiss and Radley (2010) used a cognitive apprenticeship approach with apparent success, but many institutions with a large intake will surely be put off by the cost estimate of "one faculty hour per student per week". Finding suitable facilities for small-group work can also be a problem.

A strongly teacher-controlled learning environment is usually easier to implement on a budget than a learner-driven method such as PBL. When specific topics are covered in a predetermined sequence, teachers can compensate for a poor teacher-student ratio by working as much as possible of their own pedagogical expertise into the learning environment – into explanatory materials, ready-made code, worked-out examples, lectures etc. – in a way that is reusable and can be prepared in advance. The cost of such preparation is not negligible but does reduce the need for expensive face-to-face guidance.

Automatic assessment

Many CS1 teachers, especially those who teach large classes, make use of computers to assess students' work and provide feedback automatically. Approaches to the automated assessment of programming assignments have been reviewed by Ala-Mutka (2005), Carter et al. (2003), and Ihantola et al. (2010).

Compared to direct guidance from human teachers, automatic marking and feedback is cheap in the long run, as development and deployment costs even out over a period of time. There are benefits beyond money, as well: an automated assessment system is quick, available 24/7, precise, consistent, and tireless. The main downside is that such tools are limited to particular kinds of closed assignments which have predictable outcomes, and to particular kinds of feedback. There are systems for assessing program functionality, certain aspects of coding style, test coverage, and/or answers to multiple-choice questions, but good feedback on important issues like design quality or the programming process cannot be fully automated (although semi-automation can help; see, e.g., Auvinen, 2009).

Despite the significant limitations, many teachers find automated assessment extremely useful in practice, and in many large courses that aim for genuine skill development, it seems to be a near-necessity. Alternative and complementary methods to automated assessment, which also reduce teacher workload, include peer and self-assessment. Each of these techniques comes with its own set of pros and cons; these are, however, beyond the scope of this thesis.

[1] There are exploitable positives to large CS1 classes too (see, e.g., Wolfman, 2002), but I focus here on certain pertinent problems.

10.2 Some approaches foster schema formation by exposing processes and patterns

The psychological literature reviewed in Chapter 4 suggests that programming expertise is to a large extent based on mental representations called schemas, which represent general information about the concepts and processes of programming. Various pedagogical strategies have been put forward that may help CS1 students form schemas.

Promoting the use of the authentic programming process as content that students should learn about is increasingly common in the CER literature (e.g., Zendler et al., 2008; Caspersen and Bennedsen, 2007; Bennedsen and Caspersen, 2008; Boisvert, 2009; Bareiss and Radley, 2010). In these approaches, the teacher's task is to show how programmers really work and think as they solve problems and carry out other tasks. Not all of this work is explicitly grounded in a cognitive perspective on learning. However, from a schema theory point of view we can interpret it as facilitating the formation of scripts and problem-solving schemas that characterize the cognitive patterns of a competent programmer.

Another family of schema-theory-based CS1 pedagogies relies on the specific algorithmic patterns which occur and reoccur in different kinds of programs, and which experts recognize and work with. The more recent work in this vein builds on Soloway's pioneering work (see Section 4.4.1). Many authors have identified and named common algorithmic schemas that are said to capture some of the same knowledge that experts have in their mental schemas. Such patterns can be taught explicitly to novice programmers to help them on the path to expertise (see, e.g., Wallingford, 1996; Proulx, 2000; Muller, 2005; Sajaniemi, n.d.; Proulx and Cashorali, 2005; de Raadt, 2008; Hu et al., 2012, and references therein). For instance, Sajaniemi's roles of variables (which I briefly discussed on p. 42) explicate standard patterns of variable use.

As we saw in Chapter 4, programmers use a combination of top-down and bottom-up strategies when writing and reading programs. These strategies make use of schemas at different levels of abstraction. Novices who initially lack any programming schemas have to build from the lower levels upwards, while experts can work more efficiently top-down, resorting to bottom-up strategies when needed. Pattern-based approaches such as Soloway's plans and Sajaniemi's roles of variables are concerned with helping novices 'put the pieces together'. Although these approaches clearly operate at a lower level than, say, design patterns (Gamma et al., 1995), they are nevertheless concerned with fairly high levels of abstraction. To apply the plans successfully, novices need to understand the pieces that they will put together. Extended practice with low-level fundamentals may be necessary to deal with the high cognitive load inherent in learning about higher-level patterns (cf. Kranch's study on p. 70 above).

In other words, novices need low-level schemas that correspond to the primitives of the notional machine they are learning to control, e.g., "what is a reference?", "how does one pass a parameter?' The next section discusses pedagogical strategies for helping students understand the notional machine. From the perspective of schema theory, those strategies can be viewed as supporting the formation of crucial low-level schemas.

10.3 Visualizations and metaphors make the notional machine tangible

Ben-Ari (2001a) advocates explicitly teaching a conceptual model of a notional machine to novice programmers so that they will form mental models of the machine that are viable for the purpose of programming. His advice to teachers, which is based on his take on constructivism (Section 6.7), includes the following.

- Explicitly present a conceptual model of the computer. Teach about this underlying model before you teach about abstractions based on it.
- Do not require students to engage in activities without viable understandings of the abstractions that those activities are based on. Specifically: delay programming exercises until students have constructed a viable mental model of the computer.
- Guide students to fix their non-viable mental models of the computer.

While delaying programming exercises is a controversial suggestion (cf. Section 10.1 above), the importance of explicitly teaching about a notional machine is much less so. In Part II, we saw that the runtime dynamics of programs and the role of the computer as an executor of programs have been widely identified as a key challenge in introductory programming education. The importance of the notional machine is supported by multiple theoretical perspectives and by plentiful empirical evidence.

A relatively popular way of teaching about a notional machine is visualization.

Why visualization?

Visualization is intuitively appealing as an educational tool. There is also considerable support for it in the literature. Any reader of educational research will sooner or later – usually sooner – come across texts exhorting the use of visualization in learning environments. It seems that nearly every learning theory has been used as a basis for recommending the use of visualization by someone or other.

Practitioners agree. Using pictures as clarification is common in teaching in textbooks and classrooms around the world, both within and outside computing education. According to one survey, most attendees of a computing education conference use visualizations of some sort in their teaching almost every day (Naps et al., 2003).

Let us briefly consider some learning-theoretical support for visualization from the traditions of research on learning that were discussed in Part II.

Mayer (1981), like many others in the literature on psychology of programming, advocates using a visualization such as that in Figure 10.1 as a conceptual model for learning about a notional machine. More generally, Mayer's (2005, 2009) theory of multimedia learning emphasizes the need to use both the visual and the auditory channels to optimize the use of working memory.

Ben-Ari (2001b) reminds us that according to (cognitive) constructivism (Chapter 6), learning is brought about by exposing learners to situations that require them to change their cognitive structures. He points out that if we assume that cognition is at least partially visual, then visualization in general and software visualization in particular ought to be effective.

Recommendations from the phenomenographic tradition (Chapter 7) tie in with the specific content of learning. From this perspective, visualization is one of the tools that can be used to highlight variation in the critical aspects of a phenomenon to be learned about. Marton and Booth (1997) describe how a visualization system for teaching Newtonian physics allows changes in perspective, and observe that "a software tool can be used to vary something that is normally taken for granted and turning it into an object of variation". They also recommend (p. 81) making the technical implementation of recursion visible in programming education in order to draw learners' attention to the self-referential nature of recursive functions. Along similar lines, I myself have previously suggested that visualization could serve to highlight critical aspects of important programming concepts such as object and variable (Sorva, 2007, 2008).

Obviously, there are also proponents of cognitive psychology, constructivism, and phenomenography who have never recommended visualization. It appears that advocacy of visualization is to some extent independent of which general learning theory one prefers.[2]

Visualizations in the classroom

Mayer (1975, 1981) experimented with a simple diagrammatic conceptual model of a computer (Figure 10.1), which he used for explaining how simple imperative programs written in a BASIC-like language work. In Mayer's studies, learning about the conceptual model transferred to better performance in tasks that require the creative transfer of knowledge about programming constructs to new situations. One experiment suggested that the effect appears when the model is taught before the language commands, but not if the order is reversed.

[2] According to some 'learning style' frameworks – which are popular but have questionable validity (Coffield et al., 2004; Pashler et al., 2008) – there are people who are 'visual learners' and prefer to learn from visualizations. Educators who subscribe to this claim differ among themselves as to whether teaching should be tailored to match each learner's style, the alternative being to attempt to broaden learners' ability to work with different kinds of materials and approaches.

Many CS1 teachers have come up with their own conceptual models of the machine, which they have taught to students on the blackboard, in textbooks, or in electronic format. I, for instance, have used bespoke PowerPoint animations to illustrate the inner workings of object-oriented example programs at two different levels of abstraction.

A consistent and concrete high-level memory model for object-oriented programming was presented by Gries and Gries. In their model (Figure 10.2), a class is analogous to a file drawer in a filing cabinet, while the manila folders in a particular drawer correspond to objects (Gries and Gries, 2002; Gries, 2008). Gries and Gries (2002) also present a graphical notation for drawing method activations within a call stack (Figure 10.3). Gries (2008) argues for a conceptual model that "rises above the computer and that is based on an analogy to which students can relate". Gries and Gries's conceptualization of classes and objects indeed operates at a level far removed from computer hardware, but nevertheless falls well within the definition of a notional machine (Section 5.4). Their diagrams serve as a conceptual model of a high-level notional machine for object-oriented programming.

How does one use a visualization? Ross (1991) commented on the traditional way:

> *The time honored approach to teaching program execution dynamics (mainly because no other approach has been available) is for the instructor to do program walkthroughs at the blackboard, pretending to be the computer while describing the effects of executing each statement in succession. As any instructor would know, this process is very error prone, slow, and not easily repeatable should questions be raised about something that occurred earlier in the walkthrough. Perhaps the worst part of this approach is that the students have no way of capturing the walkthrough on paper for later review; if they take notes, they again wind up with a static, rather messy picture of the status of the walkthrough as it last appeared on the blackboard, with little chance of being able to repeat the walkthrough on their own later.*

One alternative to the chalk-and-talk approach is to involve the students.

Student-drawn visualizations

Having students draw the visualizations themselves is a cognitively engaging way to visualize the notional machine. Some teachers set the ability to draw program state as an explicit goal:

> *We believe that students in the first two programming courses (using Java) should have practice with a model of execution. They should be able to draw instances of classes and subclasses, with enough technical detail to determine the variable or method that is referenced by an identifier within a method body. They should be able to execute method calls by hand, including pushing the frame for a call on the call stack and popping it off when the call is completed. This material is often taught in a later programming language course; we believe it belongs in the first programming course.* (Gries and Gries, 2002)

Gries and Gries made sure their students could not simply ignore the conceptual model presented to them: "We draw objects often, and we force our students to draw them" (Gries, 2008). Assuming that having students draw is useful, making the activity obligatory may well be a sensible policy, given the empirical evidence suggesting that many novices do not voluntarily trace program execution, or draw status representations, even when it would benefit them or when they are encouraged to do so (Section 5.5).

Holliday and Luginbuhl (2003, 2004) used a similar approach to the Grieses. They introduced students to a memory diagram notation for representing program states, and had students use it in class. Student-drawn diagrams were further used as a part of assessment. Holliday and Luginbuhl report that the ability of students to draw diagrams in an exam correlated positively with their performance on other questions. It is unclear, however, if drawing memory diagrams *caused* the students to do better overall. Vagianou (2006) and Mselle (2011) have also reported on the use of memory diagrams drawn by teachers and students alike. The latter got a positive result in an experiment, which did not, however, have the same teacher in the control and treatment groups.

A kinesthetic alternative to drawing visualizations is in-class role-playing of the actions that the computer takes when executing a program. This can be done at a different levels of abstraction, in terms of object interaction, for instance (see, e.g., Andrianoff and Levine, 2002).

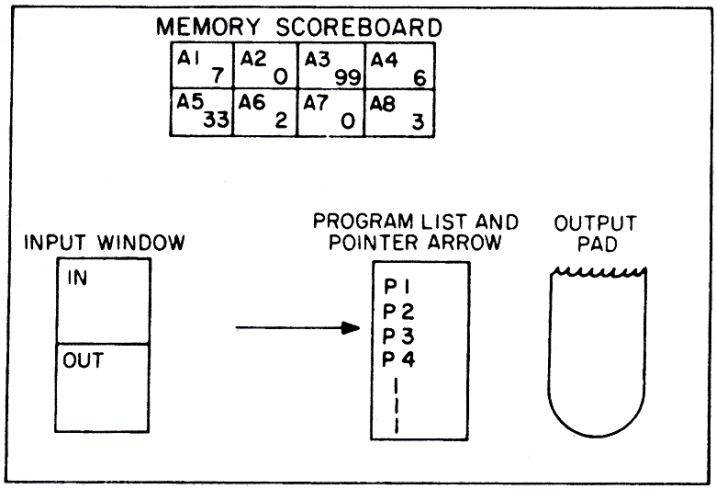


Figure 10.1: A visualization of a simple notional machine by Mayer (1981).

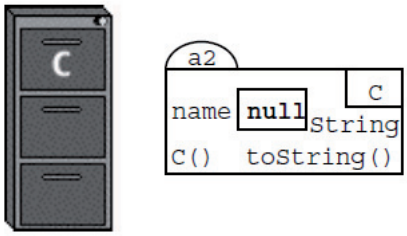


Figure 10.2: A visualization of classes and objects by Gries and Gries (2002). `C` is a class, represented as a file drawer. `a2` is an identifier for an object of type `C`, represented as a manila folder.

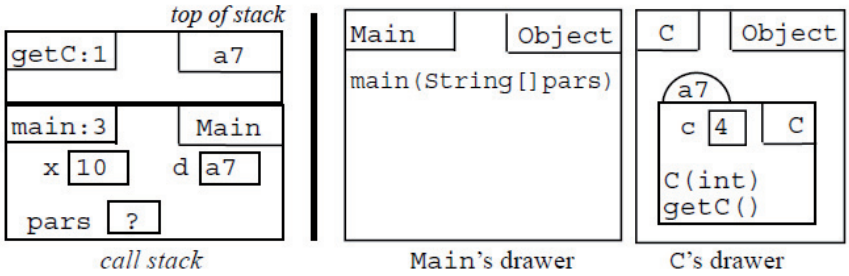


Figure 10.3: A visualization of objects and call stack frames by Gries and Gries (2002). On the right, there are two classes: `Main`, which has a static method `main`, and `C`. One object of type `C` has been created; it has the identifier a7. On the left is the call stack. In this example, the parameterless method `getC` has been called on the object a7, which is referenced by the local variable `d` of the `main` method.

Vagianou (2006) observes that a weakness of pen-on-paper methods (and comparable computer-aided drawing) is that producing pictures of program states may take up a lot of space and can be inefficient. The time needed to draw diagrams is a concern, especially if students are not convinced that that time will be well spent. The last point was also made by Gries et al. (2005), who observed that "students often complain about these exercises, as drawing (and re-drawing) diagrams is tedious". Software can make things easier. Visualization software has been used to illustrate particular programming concepts and entire notional machines.

Visualizing specific concepts with software

Many CS1 teachers employ visualizations of specific programming concepts and parts of a notional machine. For instance, to teach about variables and references to objects, I myself have used an analogy that likens variables to pegs, and objects to balloons that are attached to pegs with strings (adapted and extended from Barker, 2000), and Bataller Mascarell (2011) has presented an assortment of visualizations to be used in CS1 under the mantra "one concept, one drawing".

Generic presentation software such as PowerPoint is one way to give students access to visualizations. Another is to use software that is custom-made for programming education. Naps and Stenglein's (1996) visualization of scope and parameter passing (Figure 10.4), Ma's (2007; Ma et al., 2011) visualization of variables and assignment, and Kumar's (2009) tutorial on pointers are examples of software tailored to visualize specific programming concepts. The WadeIn II system (Brusilovsky and Loboda, 2006) visualizes stages of expression evaluation and quizzes students about them, as does another "problet" by Kumar (2005). The BlueJ IDE's object workbench – shown in Figure 10.5 – allows students to experiment with objects (Kölling, 2008); a similar functionality is provided by Aguia/J (Santos, 2011). Anchor Garden (Miura et al., 2009) facilitates the visual exploration of the runtime semantics of variables, references and assignment: the system serves as a sort of exploratorium for these concepts (Figure 10.6). Recursion is another topic for which several specialized visualizations have been proposed (see, e.g., Eskola and Tarhio, 2002; Velázquez-Iturbide et al., 2008, and references therein).

The Python Visual Sandbox (Weigend, n.d.) represents an unusual approach: it is a web site which shows visiting students multiple alternative visualizations of the internal behavior of programming constructs such as list assignment and recursion – most of them at a high level of abstraction (Figure 10.7). The visitor is then invited to think about which of the visualizations accurately represents the runtime behavior of those constructs.

Generic program visualization software

Generic program visualization systems take as input a program written in a real programming language (or a subset), and provide a dynamic visualization of how a notional machine deals with the program. A familiar example is a common visual debugger in any modern IDE, which shows the execution of a program line by line, and displays some pertinent information about the contents of computer memory, most importantly variable values and call stack frames. More learning-oriented and beginner-friendly systems also exist. Jeliot 3 (Moreno et al., 2004), for instance, explains program execution in a detailed fashion using a visualization of a notional machine for Java programming.

Generic program visualization systems are central to this thesis. I will discuss existing systems in more detail in the next chapter. Later, Chapter 13 introduces a new generic program visualization system for CS1.

Warning: difficulties in transfer

The literature widely suggests that learning about a notional machine is important and that the ability to trace programs is eminently useful. However, there is also some evidence that explicit teaching about how programs get executed does not always lead to transferable knowledge.

Studies have indicated that novices often depend on templates and analogies to write recursive programs (see, e.g., Kessler and Anderson, 1986; Anderson et al., 1988). According to Pirolli (1991), producing a recursive function is, in principle, a "rather straightforward" task in which there is "little

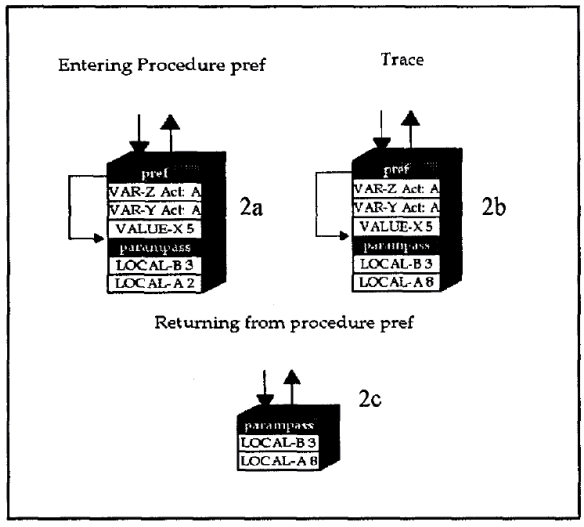


Figure 10.4: A visualization of the stack during a Pascal procedure call (Naps and Stenglein, 1996). A procedure of the form `pref(x : integer; var y, z : integer)` is called with the expression `pref(a+b, a, a)`. `pref` modifies the value of a (full code not shown here).

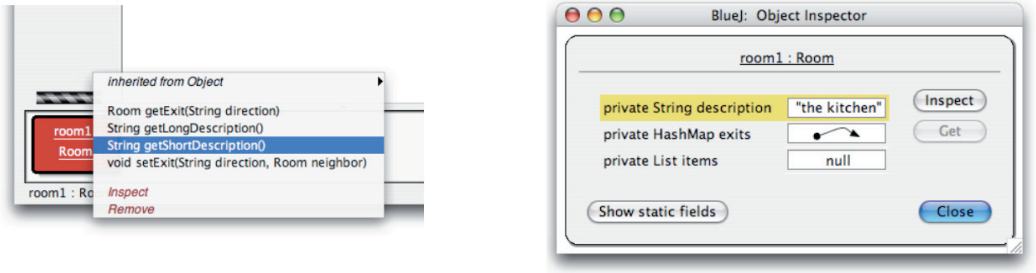


Figure 10.5: Images from the BlueJ IDE (Kölling, 2008). The image on the left shows a context menu that the user has opened by clicking on an object in BlueJ's object workbench. The user can invoke the object's methods through the menu. The image on the right shows an object inspection dialog that displays the object's state.

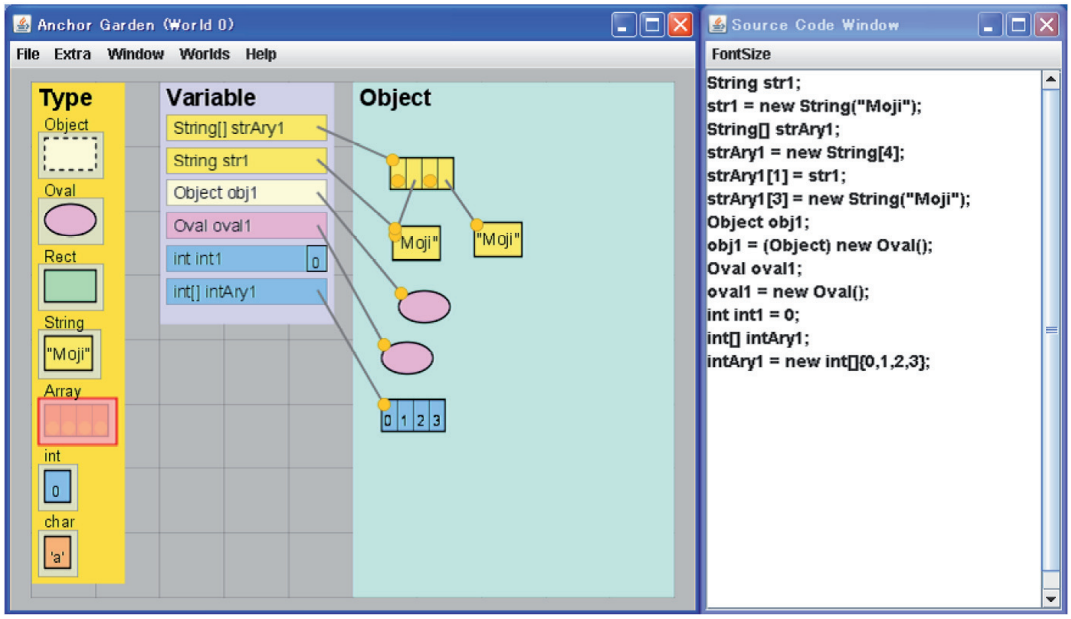


Figure 10.6: A view of Anchor Garden (Miura et al., 2009). The user can use the palette of tools on the left to create variables, objects, and primitive values. Variables can be assigned to by using the mouse. In the "advanced mode" (shown), the fruits of the gardener's labor are shown as Java code on the right.

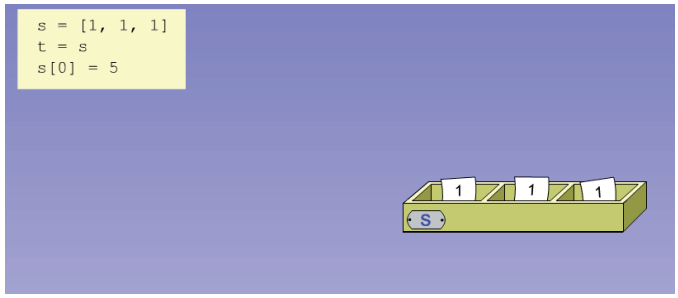


Figure 10.7: One of the alternative visualizations of lists and variables shown in the Python Visual Sandbox (Weigend, n.d.). Note that this visualization of the code appears to store the names of referring variables in the list object. This makes it a questionable visualization for the concept, which the user is expected to detect.

or no need for simulating the process created by a recursive function". Pirolli argues that the main difficulty comes from a lack of analogues for recursion in everyday thinking. If this is the case, it is perhaps reasonable to expect the skill of writing recursive programs and the skill of tracing them to be only loosely connected to each other. Studies by Anderson, Pirolli, and their colleagues have indeed found that teaching students about the execution of functional (typically recursive) programs can help students develop the skill of tracing, but that this skill does not necessarily translate into an ability to write programs. Anderson et al. (1989) review experiments from their research group, which indicate that training students to evaluate LISP expressions manually (i.e., to simulate execution within a LISP notional machine) did not significantly help them to write LISP programs. Conversely, training in coding transferred rather poorly to expression-evaluation skill. On the basis of an analysis of exam questions, Anderson et al. (1989) further suggest that evaluation training transferred poorly even to debugging skill. They do not report the exam questions they used, however, and it is unclear to what extent the debugging questions in the exam necessitated the tracing of execution. In another study, Pirolli (1991) experimented with two ways of teaching recursion by giving some students "how-it's-written" information (about how to put together a recursive program) and others "how-it-works" information (about how a recursive program behaves when run; in other words, about a notional machine). He found that how-it's-written information was more effective in teaching the learners to write small recursive programs, producing equal success in less time.

Visualizations of notional machines are all about how-it-works information. The findings cited above contrast somewhat with the success stories reported from within the software visualization field (see the next chapter). It is not obvious to what extent the limitedness of transfer discovered by Anderson, Pirolli, and their colleagues is particular to recursive programs, LISP, or the functional paradigm and its straightforward high-level notional machine. Even within the realm of LISP programming, positive transfer effects from how-it-works information to debugging have also been documented through empirical research (Mann et al., 1994).

The alternative: leaving the notional machine implicit

By no means all CS1 teachers explicitly teach their students about a notional machine. Not everyone agrees that it even makes sense to do so. For instance, Greening (1999) downplays the need to teach a model of a computer (see Section 6.7). He claims that with the employment constructivist practices such as PBL (see Sections 6.4 and 10.1), problems will not arise because students will necessarily discover viable understandings of the computer while working on larger projects: "a constructivist environment would not find students writing the trivial code fragments needed to allow such misconceptions to escape" (p. 74). If this is the case, the implications for introductory programming education are significant; however, the CER literature presently appears to provide little in the way of concrete evidence supporting this argument.

Many textbooks appear to neglect the issue of a notional machine. It is not rare that the computer is introduced in terms of its hardware and perhaps machine code, but that that low level of abstraction is only very loosely linked to the execution-time dynamics of a higher-level language.

An interesting observation was made by Schulte and Bennedsen (2006), who surveyed the opinions of programming teachers on the relative importance of various CS1 topics. They report that even though programming teachers found some specific dynamics-related topics (e.g., references) to be relatively important compared to other specific topics, the notional machine itself was seen as relatively unimportant compared to learning about notation and pragmatics, among other things. Schulte and Bennedsen's result may be affected by the teachers not agreeing on what exactly is meant by "notional machine", but probably also reflects the status quo in CS1 teaching, in which the big picture of how programs work at runtime does not get quite the attention it deserves in the light of learning theory and empirical evidence.

10.4 Objects are a bone of contention

A chapter on teaching CS1 would not be complete without a consideration of the so-called objects-early debate.

It is a common sentiment that the first few weeks of a CS1 course are critical for student success. First

impressions of computing matter and affect student motivation; what is learned first may set the tone for the rest of the course and color further learning. Some, plausibly many, CS1 strugglers start experiencing difficulties already near the beginning of the course (Teague et al., 2012). The tight integration of programming topics may mean that failing to learn what is covered first leads to failure to learn further concepts (Robins, 2010). Key threshold concepts such as program dynamics may need to be mastered right at the start (Chapter 9). For these reasons, various 'this-first' and 'that-early' pedagogies have been proposed, each of which emphasizes the importance of teaching about a particular topic at or near the start of CS1: objects first, objects early, functional first, imperative first, design patterns first, components first, testing first, and so forth.[3] An ongoing – and often heated – debate amongst practitioners concerns whether to start out with object-oriented or procedural programming (Bruce, 2004; Lister et al., 2006a). Since the 1990s, objects-early approaches have become common, but more traditional procedural-first approaches also continue to be popular. I will gloss over functional-first approaches, which have not been focal in this debate; for an example of a relatively popular functional-first approach, see Felleisen et al. (n.d.).

Lewis (2000) observed that the term 'objects first' is commonly bandied about with no clear definition and that a lot of significantly different approaches are lumped together under that general term. Bennedsen and Schulte (2007) analyzed descriptions of 'objects first' elicited from over 200 teachers from the world over, and found three main meanings for the term: 1) using objects of predefined classes right at the beginning at CS1; 2) defining and instantiating classes right at the beginning, and 3) learning about general OO principles and object-oriented modeling right from the start. At the risk of oversimplifying, I will consider all of these approaches as falling under the term *objects early* below, while *objects later* will refer to all approaches that do not emphasize object-orientation in the early weeks of CS1.

10.4.1 Is OOP too complex to start with?

> *One wonders* [...] *about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops; more time spent on looping problems might pay a much larger return in the long run.* (Winslow, 1996)

Winslow's sentiment has been echoed by educators who oppose the objects-early movement and prefer to stick to (or return to) procedural or functional programming for all or most of CS1. The reasoning is that object-orientation brings further content into an already content-packed CS1 and 'basic constructs' such as loops and conditionals will not get the necessary attention. It has also been argued that such basics must logically be taught very early, which is not always done in objects-early approaches (Bruce, 2004; Lister et al., 2006a). A prominent critic of objects-early is Stuart Reges, who is unconvinced that many students are capable of learning about OOP early, that many teachers have what it takes to successfully teach about objects early, and that, in general, objects-early approaches solve more problems than they create (Bruce, 2004; Reges, 2006).

Sajaniemi and Kuittinen (2008) revisit the objects-early debate from the notional machine point of view, warning us that object-oriented programming inevitably requires a more complex notional machine than procedural programming does (see Section 10.4.3). This, Sajaniemi and Kuittinen argue, is prone to cognitively overloading novices, something that is easier to avoid when using a procedural approach. Critics of objects early have also pointed to the large number of different misconceptions that novice programmers have of object-oriented concepts (see Appendix A). Ben-Ari (2001a), who advised us in Section 6.7.2 that learning computing should not start with abstractions, argues against objects early by noting that a high level of abstraction is inherent in OOP, and misconceptions are inevitable if students start with OOP without being taught what lies beneath. The claim that OOP is a 'natural' way of programming has also been questioned (Pane et al., 2001; Guzdial, 2008).

Reges (quoted by Bruce, 2004) argues that the existence of various elaborate forms of scaffolding (see below) is an indication of how OOP is unnecessarily complicated as a starting point. He suggests that OOP may not be very fundamental to computing at all. Ben-Ari (2010) agrees with the latter point as he presents one of the more radical criticisms against objects early. According to Ben-Ari, object-oriented

[3] There is even a "pigs-early" approach (Lister, 2004). Some of the initiatives that address gender balance issues in CS1 (e.g., Rich et al., 2004) might be termed 'ladies first'.

programming is not a dominant technology, and OOP advocates have not done a good job of explaining exactly what OOP is good for and why it is a good idea to teach it to all programmers, let alone to start off beginners with objects.

10.4.2 Is OOP too good not to start with?

Object-oriented programming holds appeal for educators for many of the same reasons that were associated in Section 10.1 with complex, authentic assignments: objects-early approaches are said to improve student motivation, to facilitate the teaching of good professional practices, and to be the natural way to start when the goal is to teach about this popular programming paradigm. Skeptics, too, acknowledge some of these benefits:

> *I appreciate the attractiveness of an objects-first approach; the gap between the standard libraries (especially the GUI libraries) of a modern programming environment and the model of a computer is so great that motivating beginners has become a serious problem. Furthermore, OOP can be used to teach good software development practice from the beginning because "OOP allows – even encourages – one to address the "big picture" by emphasizing a* strategic *approach to programming" (Decker and Hirshfield, 1993, p. 271).* (Ben-Ari, 2001a)

It is for such reasons that proponents of objects early feel OOP is a suitable choice for the modern CS1 course, despite the challenges it poses for teachers (see also Bruce, 2004).

Goodbye, Hello!

Compared to its alternatives, a particular benefit of an objects-early approach is said to be how it allows teachers and students to rid themselves of "Hello, World!" and other trivial early programs, whose behavior is uninteresting, useless, and unmotivating. Many authors have stressed the limitations of such examples and the downright harmful effects they may have on learning (e.g., Westfall, 2001; Dodani, 2003; Kölling, 2008). In many objects-early courses, by contrast, early examples feature classes with rich behaviors, whose inner workings students may not be at all familiar with at the beginning.[4] A number of authors have emphasized that in order to keep misconceptions to a minimum, early object-oriented examples must be particularly carefully designed and crafted by the teacher (e.g., Holland et al., 1997; Caspersen and Bennedsen, 2007; Nordström and Börstler, 2011).

Scaffold to succeed

Objects early is claimed to allow natural treatment of larger, more interesting problems. In fact, an object-oriented CS1 probably requires such examples in order for students to appreciate the power of object-orientation. Objects-early teachers use various scaffolding techniques to manage complexity. Rasala (quoted by Bruce, 2004) groups these methods in three categories:

- *pedagogical IDEs* such as BlueJ (Kölling et al., 2003; Kölling, 2008) and DrJava (Allen et al., 2002), which allow novices to easily create and interact with interesting objects without writing a full program;
- *microworlds* such as Alice (Cooper et al., 2003) and Karel J. Robot (Bergin et al., 2005), which feature ready-made classes that provide graphics and rich primitives for students to work with;
- *pedagogically designed class libraries* such as ACM's Java library (Roberts et al., 2006) and Java Power Tools (Rasala et al., 2001), which provide novice-friendly APIs for various purposes, especially for creating graphics and GUIs.

A further consideration is the way topics are sequenced, which can be designed to minimize the risk of cognitive overload, as in Caspersen and Bennedsen's (2007) consume-before-produce CS1 discussed above in Section 10.1.

[4] OOP does not, of course, have a monopoly on rich behaviors hidden behind interfaces. However, starting with such examples is common in object-early approaches, and relatively uncommon otherwise.

The paradigm shift

There is anecdotal evidence both for and against the claim that – assuming one's eventual goal is to teach OOP – it makes sense to start with object-oriented programming rather than to require students to shift from the procedural paradigm to OOP. The latter alternative requires students to change their way of thinking about problems and their programming solutions and is said to involve difficult 'unlearning' processes.

> *This is why I think it is a mistake for educators to try to teach procedural programming first, if the goal is object programming. The better a job you do in the beginning, the harder it will be for your students later, because their natural problem solving skills will all be wired to look for solutions in the processes and not in the objects. If you do a really excellent job of teaching them to think like a procedural programmer, they will face this 12 to 18 month paradigm shift. I don't know where to put this year of confusion in a four-year educational program.* (Bergin, 2000)

Strong evidence both for and against such claims is scarce, as is direct evidence about the effectiveness of objects early in general (see Robins et al., 2003; Decker, 2003; Lister et al., 2006a; Ehlert and Schulte, 2009, 2010). Ehlert and Schulte (2009, 2010) compared the results of an objects-first CS1 and an objects-later one, which were otherwise identical to each other (same teacher, etc.). They report that "both groups showed the same increase in learning gain, but perceived the difficulty of topics differently". Earlier, Wiedenbeck et al. (1999) reported that novices trained in procedural programming outperformed object-oriented novices in program comprehension questions (when the programs were not very short). Wiedenbeck et al. speculated that this was due to the fact that object-oriented programming is more complex and has a longer learning curve. Consequently, the object-oriented novices who had studied programming for an equally long time had not made as much progress as a result of having had to cope with additional topics in their studies.

10.4.3 OOP can be expressed in terms of different notional machines

To follow the advice given in the previous sections and chapters, an objects-early teacher should provide a conceptual model of a notional machine that explains the dynamic behavior of object-oriented programs at a reasonable level of abstraction. But what is an appropriate object-oriented notional machine like?

An extended imperative machine?

Some have argued that a notional machine suitable for object-oriented programming is an extension of a procedural or imperative notional machine. Sajaniemi and Kuittinen (2008) compare two notional machines for object-oriented programming and procedural programming, both of which describe execution at the same level of abstraction. They argue that a notional machine for very simple imperative programs needs to feature (only) variables, I/O devices, and a program counter. It can be seamlessly expanded into a richer notional machine by adding pointers, a call stack, and mechanisms for parameter passing and returning values once students reach programs involving pointers and functions. In contrast, Sajaniemi and Kuittinen argue, any object-oriented notional machine must be more complex, featuring objects, object references, a call stack, mechanisms for parameter passing and return values, variables, I/O devices, and a program counter. According to Sajaniemi and Kuittinen, "not only is the size of the required notional machine much larger than in the procedural case, but the initial notional machine needed in order to understand the first programs is much more complicated, as well". They draw on the literature to further claim that "the OO notional machine is even more poorly understood by students than the imperative notional machine".

Schulte and Bennedsen (2006) briefly mention an object-oriented notional machine that "comprises traditional imperative aspects of the programming language as well as an understanding of the interaction among objects that take place during run-time". This phrasing suggests an object-oriented notional machine that is an extension of an imperative one, but also involves higher-level aspects.

A high-level OO machine?

Sajaniemi and Kuittinen's object-oriented notional machine operates on the same level of abstraction as their procedural notional machine, dealing with primitives such as variables and stack frames. In contrast, Bergin emphasizes the dramatically different way computation is thought about in OOP:

> *One fairly typical component of a beginning course of programming using the procedural paradigm is a discussion of the von Neumann machine architecture.* [...] *This simple machine model fits well with the procedural paradigm, but less well with other, more abstract, ways of looking at computation. There is a simple relationship between the physical level provided by the von Neumann architecture and the virtual level provided by most procedural languages. This is just not the case with the functional or object-oriented paradigm. The functional paradigm, of course, completely hides the underlying physical architecture. The object-oriented one does not hide it, but turns it on its head. Instead of the data being moved to the CPU for processing, a very common metaphor in OOP is that the CPU moves inside the objects.* (Bergin, 2000)

Gries (2008) also criticizes the use of low-level abstractions in teaching object-orientation:

> *But many programming texts fail to use abstraction appropriately, e.g., by describing variables and assignment in terms of computers* [...]
>
> > *"A variable is a name for a memory location used to hold a value of some particular data type."*
> >
> > *"When [the assignment statement is] executed, the expression is evaluated . . . and the result is stored in the memory location . . . ."*
> >
> > *"The computer must always know the type of value to be stored in the memory location associated with a variable."*
> >
> > *"An object reference variable actually stores the address where the object is stored in memory."*
>
> [...] *introducing computing concepts in terms of the computer can create unnecessary and confusing detail, especially when OO concepts are described in terms of a computer, with discussions of pointers to objects in memory, heaps, and other implementation-related terms.* (p. 32)

Gries prefers to scrap difficult terminology – e.g., 'reference', 'pointer' – as "with appropriate abstraction away from the computer, these terms become unnecessary". We saw Gries's higher-level conceptual model of objects and classes in Section 10.3.

In a similar vein to Gries, Caspersen and his colleagues (Caspersen, 2007; Bennedsen and Schulte, 2006; Henriksen, 2007) have sought to represent an object-oriented notional machine on a higher level of abstraction. They argue that object-oriented programmers need to understand program execution in terms of a notional machine that deals with interacting object structures. Henriksen (2007) envisioned a visualization of a notional machine that would abstract out execution details such as expression evaluation and concentrate on object interactions. Caspersen (2007) outlines a future system in which students could 'play' with the visualization of an object model, stepping forward and backward, and making changes to program state at will (cf. Victor, 2012).

Two machines?

Berglund and Lister (2007, 2010) found through phenomenographic analysis that amongst participants in the objects-early debate, objects early is experienced in different ways: as learning an extension of imperative programming or as learning something conceptually quite distinct from imperative programming. This qualitative divide is, in my interpretation, reflected in the literature on object-oriented

notional machines. Sajaniemi and Kuittinen's OO notional machine is an example of the former perception, while Bergin represents the other line of thinking.[5]

Most writers, perhaps for simplicity's sake, talk of a single procedural/imperative notional machine and of a single, more complex, object-oriented notional machine. Another way to think about the matter, which Henriksen (2007) alludes to and which I prefer, is that while a single notional machine may be enough to understand procedural/imperative programs, object-oriented programming effectively requires (at least) two different notional machines. One can be seen as an object-enabled extension of a procedural notional machine à la Sajaniemi and Kuittinen, and another describes message-passing between interacting objects. The two notional machines operate on different levels of abstraction and give two different perspectives on object-oriented programming. This is consistent with the idea that object-oriented programming is simultaneously an extension of imperative programming and something conceptually different from it.[6]

10.4.4 So who is right?

Much of the objects-early debate is driven by anecdotes and attitudes, and is not very well grounded in research literature. Irrespective of what credibility one attaches to these opinions, they unquestionably form a part of the CER community and have a great impact on how CS1 courses are being taught around the world.

I present here some of my own opinions, based on the above and my experience as a CS1 teacher.

There are strong arguments both for and against objects-early approaches. It seems clear that both objects-early and objects-later approaches *can* work, under the right circumstances, if they are carefully implemented.

The existing arsenal of educational tools available for teaching OOP, the perceived need for these tools, and some empirical research all support the idea that object-oriented programming is 'more' than imperative programming from a learning point of view. This makes objects-early more difficult to teach successfully – more is demanded of the teacher so as not to demand too much of the students. Even more scaffolding is needed to teach an object-oriented CS1 than in an imperative or objects-later one. If you succeed, however, you have accomplished more.

A significant challenge with objects-early is that just as OOP has the dual nature of being an extension of imperative programming and a separate paradigm in its own right, so novice students need to learn about two different notional machines. One of these can be thought of as an extension of an imperative notional machine and the other as a more abstract, purely object-oriented machine. My feeling is that objects-early students need – or at least can greatly benefit from – explicitly being taught about both notional machines early. This adds further to the object-oriented teaching challenge.

In this chapter, we have seen some strategies for teaching CS1. We have explored, in broad terms, themes such as complexity vs. manageability, the role of object-orientation, and the importance of notional machines. In the next chapter, we take a narrower but deeper look at programming pedagogy to find out what software tools have been created to visualize program dynamics for novice programmers.

[5] OOP can also be seen as an extension of functional programming, but this perspective has, perhaps surprisingly, only rarely featured in the objects-early debate. I focus here on the mainstream, imperatively grounded OOP that is commonly taught in introductory courses.

[6] Whether one agrees that this statement about OOP is true depends, of course, on one's definition of OOP. I am talking here of OOP in the form in which it appears in programming languages such as Java, Python, and C++, in which it makes sense to think of OOP as (also) an extension of imperative programming.

Chapter 11

Software Tools can Visualize Program Dynamics

Software visualization (SV) is an active field of research and system development. A wide variety of SV systems exist for many different purposes and more crop up every year. In this chapter, I review software visualization tools for teaching introductory programming.

Gračanin et al. (2005) define software visualization as follows:

> *The field of software visualization (SV) investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code), and processed data. SV is concerned primarily with the analysis of programs and their development. The goal is to improve our understanding of inherently invisible and intangible software* [...] *The main challenge is to find effective mappings from different software aspects to graphical representations using visual metaphors.* (p. 221)

Software visualization is a fairly broad field in which educational concerns play only a small part. In Section 11.1 below, I use existing classification systems to specify what kind of SV tool I am presently interested in. To provide an additional context for what comes later, Section 11.2 surveys a current debate in the educational SV community that concerns the importance of learners actively engaging with visualizations, and presents a framework that I have used for classifying modes of engagement in the tools that I review. The lengthy Section 11.3 contains the review proper: descriptions of specific visualization systems and their empirical evaluations. Finally, in Section 11.4, I briefly reflect on what is known about the role of engagement in the systems reviewed; it turns out that there is rather little in the way of empirical evidence.

Ville Karavirta and Lauri Malmi participated in reviewing the systems described in this chapter.

11.1 There are many kinds of software visualization tools

As the field of software visualization is varied, many authors have tried to structure it by proposing classification systems and taxonomies of software visualization tools (see, e.g., Myers, 1990; Price et al., 1993; Stasko and Patterson, 1992; Roman and Cox, 1993; Maletic et al., 2002; Hundhausen et al., 2002; Naps et al., 2003; Lahtinen and Ahoniemi, 2005, and references therein).[1] From a different perspective, Kelleher and Pausch (2005) laid out a classification of software environments for use in introductory programming education; the set of these systems overlaps with SV. In this section, I will use the classification systems of Maletic et al. (2002) and Kelleher and Pausch (2005) to give an overall feel for the field and to specify the scope of my review of software visualization tools in Section 11.3 below. Let us start, however, by considering some broad areas within software visualization.

[1] My use of the terms "taxonomy" and "classification" (or "classification system") – which are often used interchangeably – is based on that of Bloom (1956, p. 17). According to Bloom, a *classification* serves a purpose if it is communicable and useful, while a *taxonomy* is intended to be validated through research. A taxonomy can also be used as a classification.

Program visualization vs. algorithm visualization vs. visual programming

Within software visualization, two broad subfields can be identified (Figure 11.1). *Algorithm visualization* (AV) systems visualize general algorithms (e.g., quicksort, binary tree operations), usually at a high level of abstraction, while *program visualization* (PV) systems visualize concrete, implemented programs, usually at a lower level.[2] Within program visualization, some visualizations represent program code (e.g., dependencies or code evolution) while others illustrate the runtime dynamics of programs. *Program animation* refers to dynamic visualization of program dynamics – the common variety of program visualization in which the computer determines what happens during a program run, and visualizes this information to the user (e.g., as in a typical visual debugger). Also within program visualization, research on *visual programming* attempts to find new ways of *specifying* programs using graphics rather than visualizing software that is otherwise in non-visual format.[3]

This chapter focuses on program visualization tools. Algorithm visualization tools operate at a level of abstraction that is too high to be interesting for learning about the fundamentals of program execution. The review presented also does not cover visual programming. Although many visual programming environments do feature a facility for animating the execution of a visual program (see, e.g., Carlisle, 2009; Scott et al., 2008), I have here focused on mainstream programming paradigms. I will, however, include a few select systems with AV and visual programming functionality, which have additional features that are intended for teaching about program dynamics of non-visual languages to novices.

Maletic et al.'s classification

In the task-oriented classification by Maletic et al. (2002), software visualization systems are primarily categorized by their purpose:

- **Tasks** – why is the visualization needed? (e.g., reverse engineering, defect location)
- **Audience** – who will use the visualization? (e.g., expert developer, team manager)
- **Target** – what is the data source to represent? (e.g., source code, execution data)
- **Representation** – how is it represented? (e.g., 2D graphs, 3D objects)
- **Medium** – where to represent the visualization? (e.g., onscreen graphics, virtual reality)

Even though Maletic et al. are actually concerned with programming-in-the-large and not with education, their framework is well suited to explaining what part of the SV landscape I am interested in. The *task* of the systems I review is to aid the learning and teaching of introductory programming, with an intended *audience* of novice programmers and introductory programming teachers. This goal is different from the goal of many other SV systems, which seek to help (expert) programmers to learn about the complex software systems that they visualize. As for the *target* dimension, my review focuses on systems that visualize the execution-time dynamics of concrete programs. Any form of *representation* will do as far as my review is concerned, as long as the *medium* is an electronic one and the visualization can be viewed onscreen.

Kelleher and Pausch's classification

Kelleher and Pausch (2005) defined a hierarchical classification of programming environments and languages for novice programmers. A part of the hierarchy is shown in Figure 11.2; I have left out some subcategories which are not central to the present work.

At the top level, Kelleher and Pausch divided environments and languages into *teaching systems*, which attempt to teach programming (as understood in the mainstream) for its own sake, and *empowering systems*, which attempt to support the use of programming in pursuit of another goal. My present interest lies on the teaching systems side.

[2] In some of the older literature especially, "program visualization" refers to what I have just called software visualization, but at present, the terms are commonly used the way I use them here.

[3] "VPS" in Figure 11.1 stands for "visual program simulation" which I will get to properly in Part IV.

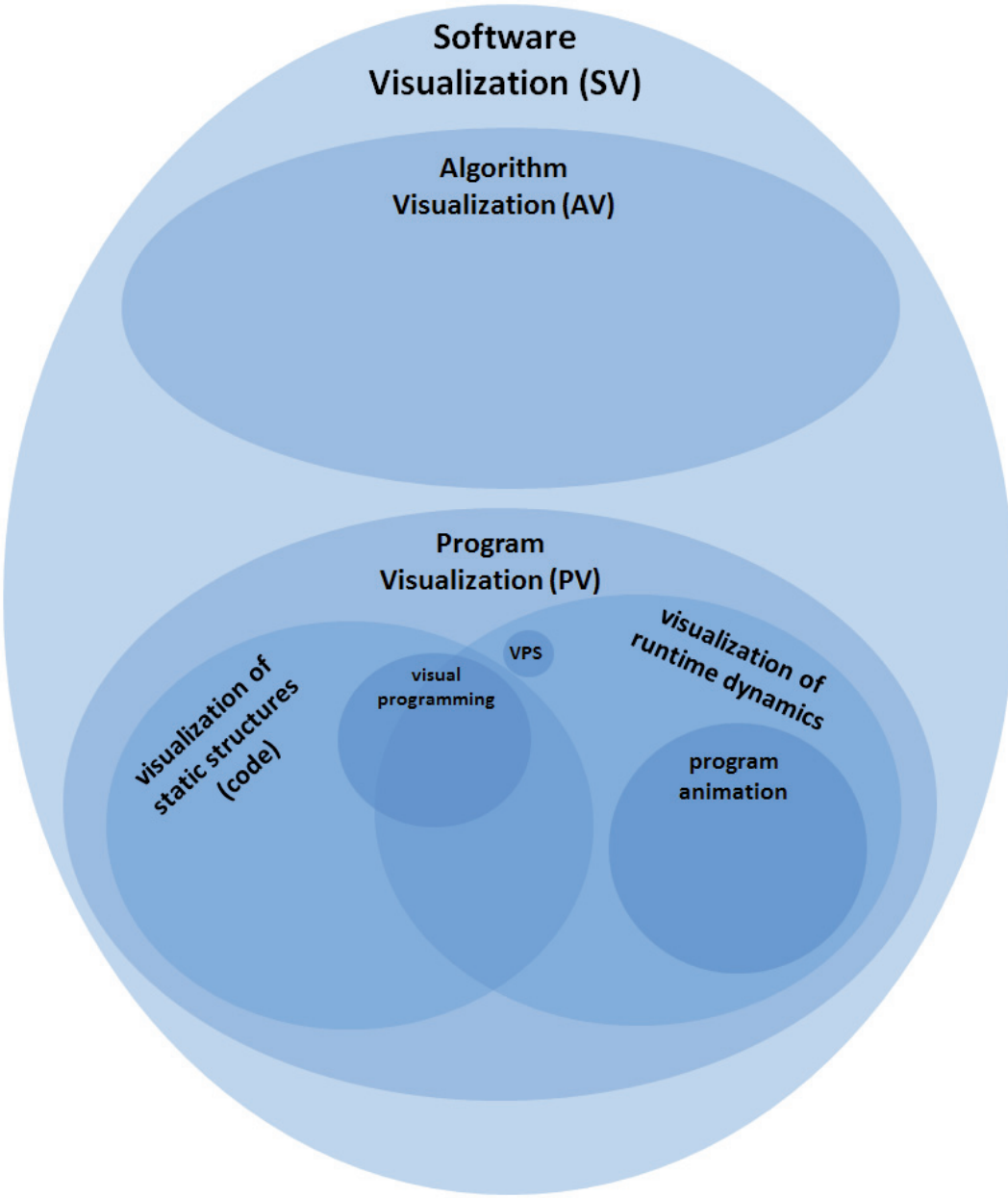


Figure 11.1: Forms of software visualization (loosely adapted from Price et al., 1993). The size of each area is not important. For the sake of simplicity, not all intersections are shown.

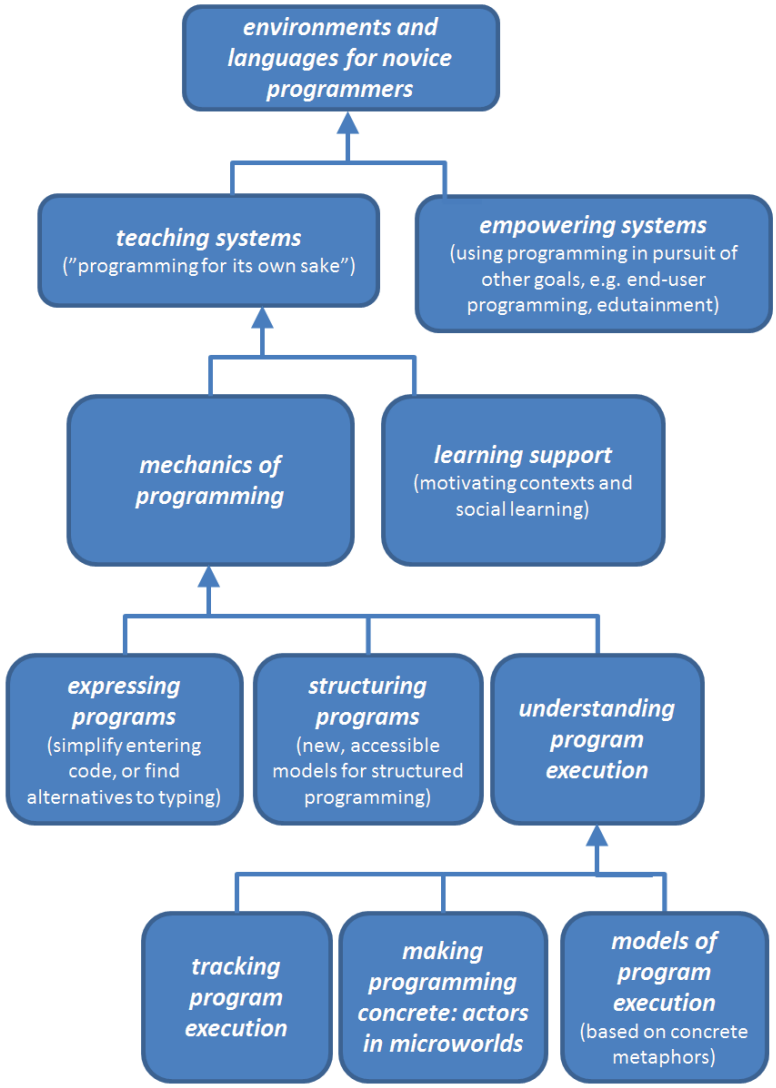


Figure 11.2: A part of Kelleher and Pausch's classification system for programming environments and languages for novice programmers. Some branches are not shown. (Adapted from Kelleher and Pausch, 2005.)

Teaching systems are further divided into *mechanics of programming* and *learning support*. Learning support systems try to ease the process of learning to program through "basic educational supports such as progressions of projects that gradually introduce new concepts or ways for students to connect with and learn from each other". Of more interest from my perspective are systems in the *mechanics of programming* category, which come in three varieties. First, systems for *expressing programs* attempt to make it easier for beginners to express instructions to the computer. Second, systems for *structuring programs* attempt to facilitate the organization of instructions by changing the language or programming paradigm (e.g., Pascal, Smalltalk) or by making existing programming languages or paradigms more accessible (e.g., BlueJ). And third, systems for *understanding program execution* attempt to help novices understand how such instructions are executed at runtime. It is systems in this last category that I intend to review.

Kelleher and Pausch identify three ways in which systems try to help students to understand program execution. A system for *tracking program execution* visualizes what happens in memory during a program run. The *actors-in-microworlds* approach makes programming concrete by replacing a general-purpose programming language by a mini-language whose commands have a straightforward physical explanation in a virtual microworld. Finally, systems in the *models of program execution* category describe actions in a (general-purpose) programming language through metaphors and graphics, which "help students both to imagine the execution of their programs and perhaps more clearly understand why their programs do not perform as expected".

As my concern is with learning general-purpose languages – which all programming students eventually need to learn and which represent the mainstream of CS1 education – I will not cover the actors-in-microworlds approach in detail. This leaves two subcategories, tracking program execution and models of program execution, both of which involve visualizations of program execution either as abstract graphics and text, or through metaphors. The line between these two subcategories is a vague one, and I will not attempt to pigeonhole systems in either of them.

I make one more delimitation: I will focus on systems that are more or less *generic* in the sense that they can be used to illustrate a variety of programming language constructs and corresponding runtime phenomena. There are also many systems which attempt to tackle more specific problems by visualizing one or a few select programming concepts. I will not attempt to cover those here. (A few examples of such specific systems were mentioned in Section 10.3.)

To summarize the above, this chapter contains a review of *generic program visualization systems* in which *the execution of programs* – written in *a general-purpose language* in a traditional non-visual way – is visualized onscreen in a manner suitable for *novice programmers* so that the system can be used to *learn about program execution*.

Before turning to the actual systems, let us consider the ways in which learners may use a visualization.

11.2 Engagement level may be key to the success of a visualization

"They will look at it and learn" thought many an enthusiastic programming teacher while putting together a visualization. But it might take more than that. Petre (1995, p. 34) writes:

> *In considering representations for programming, the concern is formalisms, not art – precision, not breadth of interpretation. The implicit model behind at least some of the claims that graphical representations are superior to textual ones is that the programmer takes in a program in the same way that a viewer takes in a painting: by standing in front of it and soaking it in, letting the eye wander from place to place, receiving a 'gestalt' impression of the whole. But one purpose of programs is to present information clearly and unambiguously. Effective use requires purposeful perusal, not the unfettered, wandering eye of the casual art viewer.*

The representational aspects of a visualization – its constituent parts and level of abstraction – are doubtless relevant to learning. In the lingo of the variation theory of learning (Section 7.3.2), these

aspects can play a decisive part in enacting a space of variation: what it is possible to learn from the visualization and what content the visualization is suitable for learning about.[4]

Yet although careful design of a visualization is important, it is not the whole story. Even a 'good visualization' may fail to aid learning in practice. In the past decade, educators with an interest in SV have increasingly paid attention to how learners *engage* with visualizations. Evidence from research suggests that what learners do with a visualization may matter a great deal, perhaps more than most of the representational issues.

11.2.1 A picture does not always better a thousand words

Naps (2005, p. 53) comments on how attitudes to software visualization in computing education have changed during the past couple of decades:

> *As often happens with eye-catching new technologies, we have gone through what in retrospect were predictable phases: An initial, almost giddy, awe at the remarkable graphic effects, followed by disenchantment that the visualizations did not achieve their desired purpose in educational applications. Then we went through a period of empirical evaluation in which we began to sort out what worked and what did not.*

The recent period of sobering reflection was fueled by a review of experiments on the effectiveness of visualization, discussed below. The original work was done on the algorithm visualization side of the SV world but the results have been applied on the PV side as well.

An influential meta-study

Hundhausen et al. (2002) conducted a meta-study of the effectiveness of algorithm visualization technology. They compared 24 earlier experiments to nutshell:

> *In sum, our meta-study suggests that AV technology is educationally effective, but not in the conventional way suggested by the old proverb 'a picture is worth 1000 words'. Rather, according to our findings, the form of the learning exercise in which AV technology is used is actually more important than the quality of the visualizations produced by AV technology.* (Hundhausen et al., 2002, p. 284)

The results from the studies were mixed overall, but the meta-study showed that the experiments which compared groups that engaged in different kinds of activities involving a visualization had produced more significant results than had those studies which compared only representational characteristics (e.g., text vs. animation or different visuals).

Engage to succeed

Hundhausen et al. (2002) observed that most of the successful AV experiments – in the sense that they showed a significant improvement in learning as a result of AV – are predicted by the theory of personal constructivism (see Section 6.1.2), which emphasizes the need to actively engage with one's environment to construct one's subjective knowledge: "our results suggest that algorithm visualizations are educationally effective insofar as they enable students to construct their own understandings of algorithms through a process of active learning" (p. 284). Hundhausen et al. showed that most of the successful AV experiments had learners engage in visualization-related activities that were cognitively more demanding and required the learner to improve their understandings of the visualization and the software it visualized. The authors concluded that visualization technology is not an instrument for transferring knowledge into students, but can serve as a catalyst for learning.

The findings of Hundhausen et al. are in line with learning theory. In Chapter 8, I observed that one of the main areas of agreement between the general learning theories discussed in Part II is their unequivocal

[4] The work of Mulholland (1997) is an example of how principled visual design driven by explicit educational goals can improve the impact of a software visualization.

support for active learning, that is, the need to make the learner engage with what is to be learned. Compatible results have also been reported by researchers of educational visualization outside SV: videos and animations can be perceived as "easy" and their impact may be inferior to that of static images or even to not viewing a visualization at all, whereas interactive multimedia that cognitively engages the learner can lead to more elaborative processing (integration with prior knowledge) and consequently to better learning (see, e.g., Najjar, 1998; Scheiter et al., 2006, and references therein).

In part, the benefit of active engagement may come simply from the fact that learners end up spending more time with the visualization, increasing the time spent on studying.

Since the study of Hundhausen et al., SV researchers have increasingly sought to create teaching approaches and systems that promote learner involvement, and to further explore how different kinds of engagement impact on learning. My thesis can be seen as part of this movement.

11.2.2 Engagement taxonomies rank kinds of learner interaction

Drawing on the work of Hundhausen and his colleagues, a sizable working group of SV researchers put their heads together to differentiate between the ways in which visualization tools engage learners (Naps et al., 2003). They presented an *engagement taxonomy* whose levels describe increasingly engaging ways of interacting with a visualization.

The original engagement taxonomy (OET)

Table 11.1 describes the six levels of the engagement taxonomy created by Naps et al. (2003), namely, no viewing, viewing, responding, changing, constructing, and presenting. Naps et al. note that the levels after viewing do not form a strict hierarchy, but they do nevertheless hypothesize that using a visualization on a higher level of the taxonomy is likely to have a greater impact on learning than using it on one of the lower levels. (In Table 11.1, and elsewhere in this chapter, I use the term *target software* to refer to the programs or algorithms that are visualized by SV software, as opposed to the SV software itself. I will sometimes also refer to the target software as the *content* of a visualization.)

SV systems sometimes feature different modes that engage students at different levels. Even a single activity may engage students at multiple levels of the taxonomy. Indeed, by definition, all activities at any but the no viewing level always involve viewing the visualization in addition to whatever else the student does. Naps et al. hypothesize that "more is better": a mix of different engagement levels leads to better learning.

The extended engagement taxonomy (EET)

Myller et al. (2009) presented a more fine-grained version of the engagement taxonomy. Their *extended engagement taxonomy*, summarized in Table 11.2, introduces four additional levels and somewhat changes the definition of some of the original categories.

The research focus of Myller et al. (2009) was on collaborative learning. They added to the hypotheses of Naps et al. by proposing that as the level of engagement between collaborating visualization users and the visualization increases, so does the communication and collaboration between the users.

Empirical support

The OET is grounded in the empirical work that inspired it, as reviewed by Hundhausen et al. (2002). Naps et al. (2003) discuss how these early experiments map onto the levels of the OET. The body of such work is not large enough to allow overall conclusions about the taxonomy to be drawn.

Since the introduction of the OET and EET, some SV researchers (mostly on the AV side) have used them to set their research questions and to interpret prior work.

Urquiza-Fuentes and Velázquez-Iturbide (2009) surveyed successful evaluations of software visualization systems in education, and relate these evaluations to the OET. They observed that various tool-supported visualization-based activities from the different levels of the OET have been found to be conducive to learning when compared to not using a visualization at all. A few evaluations also support the claims that students engaged in active forms of learning on the changing, responding, constructing,

Table 11.1: The original engagement taxonomy (OET) (based on Naps et al., 2003).

#	Level	Description
1	No viewing	The learner does not view a visualization at all.
2	Viewing	The learner views a visualization.
3	Responding	The learner responds to questions about the target software.
4	Changing	The learner changes a visualization, e.g., by changing the input given to the target software.
5	Constructing	The learner constructs a visualization, e.g., by drawing, by combining visual elements provided, by directly manipulating a predefined visualization, or by annotating source code to produce a visualization as the program is executed.
6	Presenting	The learner explains a visualization to others.

Table 11.2: The extended engagement taxonomy (EET) (based on Myller et al., 2009).

#	Level	Description
1	No viewing	The learner does not view a visualization at all.
2	Viewing	The learner views a visualization without interacting with it by other means.
3	Controlled viewing	The learner controls how he views a visualization, e.g., by changing the animation speed or choosing which objects to examine.
4	Entering input	The learner enters input to the target software before or during execution.
5	Responding	The learner responds to questions about the target software.
6	Changing	The learner changes a visualization while viewing, e.g., via direct manipulation of the visualization's components.
7	Modifying	The learner modifies a visualization before viewing, e.g., by changing the target software or an input set.
8	Constructing	The learner constructs a visualization interactively from components such as text and geometric shapes.
9	Presenting	The learner presents a visualization to others.
10	Reviewing	The learner views the visualization in order to provide feedback to others about the visualization or the target software.

and presenting levels of the OET outperformed students who merely viewed a visualization. The relative impact of the higher levels has been little studied within CER, however. For more information on the evaluations covered by Urquiza-Fuentes and Velázquez-Iturbide's survey, see their article and references therein.

Lauer (2006) used the OET to compare the performance of three groups of students using the MA&DA AV system for viewing, changing, and constructing visualizations, respectively. He found no significant differences between the groups' performance, which he suggests may be due to the interfering effects of other factors in the experimental setup.

Myller, Korhonen, and Laakso have compared the controlled viewing and changing levels in the EET in the TRAKLA2 AV system. Myller et al. (2007b) found no significant difference between a group that used the system in a controlled viewing mode and another group which engaged with the tool on the changing level as well. The authors conclude, however, that "students without previous knowledge seem to gain more from using visualizations on [a] higher engagement level". Continuing the study, Laakso et al. (2009) also found no statistically significant differences between a controlled viewing group and a changing group. They report that this may be in part due to the fact that the students in the changing group did not use the tool as intended. Korhonen et al. (2009b) found that when AV users who work in collaboration are engaged on a higher level of the EET, they also communicate more and discuss the topic of the lesson on more levels of abstraction than when engaged on a lower level. Myller et al. (2009) got similar results with the Jeliot 3 PV system.

To summarize, it can be said that empirical work to date does tentatively support various claims behind the engagement taxonomies, but a solid general validation of the taxonomies does not exist at present. In Section 11.4, I will return to what is known about learner engagement in the specific context of the PV tools for visualizing notional machines that are the focus of my review.

11.2.3 I prefer a new two-dimensional taxonomy for describing modes of interaction

In Section 11.3 below, one of the aspects that I describe for each of the tools that I review is its support for different modes of learner engagement. I might have used the OET or EET for this purpose, but have decided instead to use my own adaptation of them. I will briefly explain the reasons for this before I outline the framework that I used.

Comments on the OET and EET

The constructing category in the OET seems crowded. Creating a visualization from scratch or from primitive components is a cognitively demanding task, which requires a kind of reflection on the properties of the visualization that is distinct in nature from what is required to manipulate a given visualization or to add annotations (of given kinds) to source code.[5]

The EET splits the activities originally placed in the changing and constructing categories variously into entering input, changing, modifying, and constructing. The reasoning behind the order of the categories in the EET is not clear, and some specific choices seem questionable. For instance, is directly changing a visualization necessarily less engaging than modifying the visualization's input set? Does entering input during viewing belong three levels lower than providing an input set before viewing (which counts as modifying)?

The EET's introduction of additional categories seems reasonable and potentially helpful. However, the EET (especially) conflates two dimensions that might best be analyzed separately: how the learner engages with the visualization itself, and the relationship that the learner has with the software being visualized.

Finally, the responding category in both taxonomies is somewhat troublesome, as there are many different kinds of questions that may be asked, some of which demand rather more cognitive effort than others. This concern does apply to other levels as well, but responding is arguably the vaguest. (Cf. the criticism of Bloom's taxonomy at the beginning of Section 2.2.)

The following is my analytical attempt to improve on the taxonomies.

[5] A similar criticism has been made by Lauer (2006).

The 2DET

The engagement taxonomy of Figure 11.3 – for lack of a better term, let us call it *2DET* – has two dimensions. The first dimension, *direct engagement*, is concerned with the engagement that the learner has with the visualization itself. The second dimension, *content ownership*, is concerned with an indirect form of engagement that results from the learner's relationship with the target software, that is, the content of the visualization. Both dimensions contribute to the learner's overall engagement with the visualization, as shown in Figure 11.3.

The levels along both dimensions are largely familiar from the OET and the EET, but I have modified some of the definitions and names. I have summarized the categories of the 2DET in Table 11.3 and will describe them in some more detail below.

The direct engagement dimension

The direct engagement dimension consists of seven categories, or levels.

On the first level, no viewing, no visualization is used. Example: reading a textual description about how an example program works.

When viewing, the learner has no control over how he views the visualization. Examples: watching a video or non-stop animation of a dynamic process without being able to control pacing, watching someone else use a visual debugger.

On the controlled viewing level, the learner plays a part in deciding what he or she sees. He may change the pace of the visualization or choose which part of a visualization to explore. Examples: stepping through a program in a visual debugger, viewing a sequence of program state diagrams printed on paper, choosing which variables to show in a visual program trace (either before or during execution), navigating an avatar through a virtual 3D world.

When responding to a visualization, the learner uses the visualization to answer questions presented to him either while or after he views it. Examples: What will the value of that variable be after the next line is executed? What is the time complexity of this program?[6]

A learner applies a visualization when he makes use of or modifies given visual components to perform some task related to the target software. The task may be carrying out some procedure or algorithm, illustrating a specific case, or creating a piece of software. Examples: simulating the stages of an algorithm using a given visualization of a data structure, using given elements to record a visualization of how the evaluation of an expression proceeds, using given visual components to produce a piece of code, annotating a program's source code so that a visualization of key stages is produced when it is run.

When engaging on the presenting level, the learner makes use of a visualization to present an analysis or description to others. What is presented can concern either the visualization or the target software. The presentation task has to be demanding enough to require significant reflection and a detailed consideration of the visualization on the part of the presenter. Examples: an in-class presentation of a visualization and the program it represents, a peer review of a visualization created by another student.[7]

Finally, creating a visualization means designing a novel way to visualize some content. Examples: drawing a visualization, producing a visualization by combining given graphical primitives previously void of meaning, writing a program that visualizes software. This creative activity represents the highest form of engagement along this dimension (cf. the revised Bloom's taxonomy from Section 2.1).

The content ownership dimension

There are four categories on the content ownership dimension.

[6] I noted above that the responding category of the OET and the EET is problematic since different questions call for different kinds of cognitive engagement. The 2DET does not fix this issue.

[7] This category effectively combines the presenting and reviewing categories of the EET. Intuitively, both appear to be tasks of analysis and communication that are roughly equally engaging. Oechsle and Morth (2007), whose work inspired the addition of reviewing into the EET, in fact used reviewing in place of – rather than on top of – the presenting level of the OET.

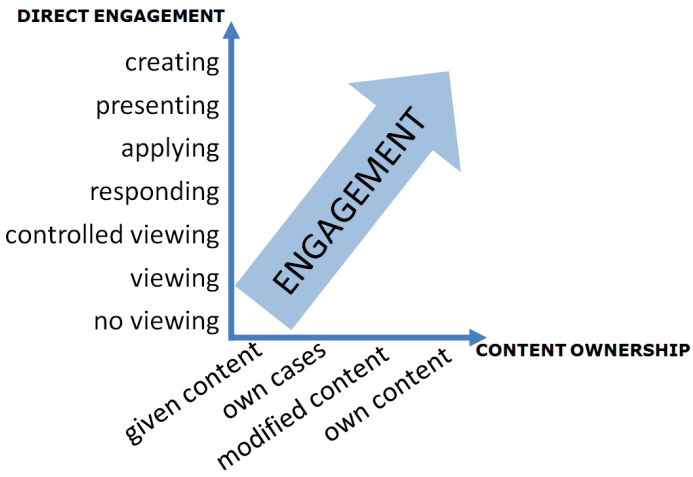


Figure 11.3: The two dimensions of the 2DET engagement taxonomy.

Table 11.3: The categories of the 2DET engagement taxonomy.

The direct engagement dimension		
#	**Level**	**Description**
1	No viewing	The learner does not view a visualization at all.
2	Viewing	The learner views a visualization with little or no control over how he does it.
3	Controlled viewing	The learner controls how he views a visualization, either before or during the viewing, e.g., by changing animation speed or choosing which images or visual elements to examine.
4	Responding	The learner responds to questions about the target software, either while or after viewing it.
5	Applying	The learner makes use of or modifies given visual components to perform some task, e.g., direct manipulation of the visualization's components.
6	Presenting	The learner uses the visualization in order to present to others a detailed analysis or description of the visualization and/or the target software.
7	Creating	The learner creates a novel way to visualize the target software, e.g., by drawing or programming, or by combining given graphical primitives.

The content ownership dimension		
#	**Level**	**Description**
1	Given content	The learner studies given software whose behavior is predefined.
2	Own cases	The learner studies given software but defines its input or other parameters either before or during execution.
3	Modified content	The learner studies given software that they can modify or have already modified.
4	Own content	The learner studies software that they created themselves.

Given content means that the learner engages with a visualization of software that they have not produced themselves and whose behavior they do not significantly affect. Examples: a teacher-given example program, a program chosen from a selection of predefined examples in an SV tool's library.

Own cases is defined as above, except that the learner can choose inputs or other parameters that significantly affect what the target software does. Example: choosing a data set for an algorithm, entering inputs that affect loop termination during a program visualization that illustrates control flow. Merely typing in some relatively arbitrary input does not count as own cases.

Modified content means that the learner engages with a visualization of given software which they have already modified themselves or may modify while using the visualization.

Own content means that the learner engages with a visualization of software that they wrote themselves.

A note on the 2DET

I have introduced the 2DET primarily to help me produce a nuanced and yet succinct and systematic description of the learning activities supported by different program visualization systems. In this chapter, I have used the taxonomy only as a classification tool for structuring my review of program visualization systems. I feel that it allows a clearer expression of modes of visualization use than the OET or the EET. I will further use the 2DET to describe the functionality of our own PV system in Part IV.

The 2DET could be used in the future in research that tests hypotheses about the role of engagement in software visualization, and could serve as a basis for a wider and more detailed review of software visualization in programming education. Such initiatives may establish how useful the taxonomy is in general, but are beyond the scope of my present work. That said, I do expect that both kinds of engagement highlighted by the two dimensions of the 2DET can help bring about the "purposeful perusal" of program visualizations that Petre called for (p. 144 above).

In the end, perhaps it matters less which classification system you use than how you engage with the one you do use.

11.3 Many existing systems teach about program dynamics

This section describes a number of program visualization tools for introductory programming education, as delimited in Section 11.1 above. A summary of the tools appears in Tables 11.5, which gives an overview of each system, and 11.6, which goes into more detail on the systems' visualization and modes of user interaction. The preceding Table 11.4 provides a legend to the other two tables.

11.3.1 Regular visual debuggers are not quite enough

> *Tools that reflect code-level aspects of program behavior, showing execution proceeding statement by statement and visualizing the stack frame and the contents of variables* [. . .] *are sometimes called* visual debuggers*, since they are directed more toward program development rather than understanding program behavior.* (Pears et al., 2007, p. 209)

Programming experts – and novices, though not as often as many programming teachers would like – use debugging software such as that shown in Figure 11.4 to find defects in programs and to become familiar with the behavior of complex software. These tools are not particularly education-oriented or beginner-friendly, but can still be useful in teaching (see, e.g., Cross et al., 2002) and may be integrated into otherwise beginner-friendly environments such as the BlueJ IDE (Kölling, 2008).

Typical visual debuggers have significant limitations from the point of view of learning programming fundamentals. They generally step through code only at the statement level, leaving most of the educationally interesting dynamics of expression evaluation implicit. The visualization and user interface controls of a 'regular visual debugger' are geared towards programming-in-the-large, not towards explicating programming concepts and principles such as assignment, function calls, parameter passing, and references, all of which the user is assumed to understand already. Only information essential for an expert programmer is shown, with the goal of helping the programmer to find interesting (defective) stages of execution in as few steps as possible while ignoring as many details as possible. The target

Table 11.4: A legend for Tables 11.5 and 11.6. In several columns of those tables, square brackets mark features that are peripheral, untried, or at a prototype stage of implementation.

Legend for Table 11.5 on p. 153	
Column	**Explanation**
System name (or author)	The name of the system. Closely related systems are grouped together as one item. Systems without a published name have the authors' names in parentheses instead.
Page	The page within this thesis where the description of the system begins.
At least since	The year when the first version was released or the first article on the system was published. May not be quite accurate, but gives an idea of when the system came into being.
Status	My understanding – or best guess – based on web searches and/or personal communication, of whether the system is still being used in teaching, maintained and/or developed. May be inaccurate.
Used for	The overall purpose of the system: what students can do with it. *Examples* is listed when the system is intended for studying given examples only. *Debugging* refers to any examination of code that can be edited within the system itself; it subsumes *examples*. *Development* subsumes debugging, and means that the system is intended to be used as a software development environment. *Assignments* means that the tool facilitates an assignment type – e.g., multiple-choice questions or program simulation – that is not a part of a programmer's usual routine of read/test/debug/modify/write/design code. Although this review does not include pure algorithm visualization systems, I mention *AV* as a goal of some AV/PV hybrid systems.
Paradigm	The programming paradigms that the software can primarily help learn about. *Imp* is used for imperative and procedural programming, which may include the occasional or implicit use of objects such as arrays. *OO* stands for richer object-oriented programming, and *func* for functional programming.
Programming language	The programming language(s) in which the programs being visualized (the target software) are written. For systems which visualize user-written programs, I have used the subscript $_{SS}$ to mean that only a significantly limited subset of a language is available for use (e.g., functions or object-orientation are missing). Lesser limitations are common and not marked.
Evaluation	The types of empirical evaluations of the system in the context of introductory programming, to the best of my knowledge. *Anecdotal* means that only anecdotal evidence has been reported about student and/or teacher experiences with the system (but this still implies that the system has been used in actual practice). *Experimental* refers to quantitative, controlled experiments or quasi-experiments. *Survey* refers to producing descriptive statistics and/or quotes from systematically collected user feedback or other data. *Qualitative* refers to rigorous qualitative research (e.g., grounded theory, phenomenography). See the main text and Table 11.7 for more details on the evaluations.

Legend for Table 11.6 on p. 154	
Column	**Explanation**
Direct engagement with visualization	The levels of direct engagement between learner and visualization which the system explicitly supports, in terms of the 2DET (Section 11.2.3). The presenting and creating levels of the 2DET do not feature in the table, as none of the systems explicitly support them – which of course does not imply that the visualizations shown by the systems cannot be presented to others. The basic engagement level of merely viewing a visualization is not listed separately unless it is the only one present (which is not the case in any of the tools reviewed). I consider each system as a generic PV system: modes of interaction that are particular to a specific subsystem or programming concept (e.g., an object inspection tool) are not listed.
Content ownership	The degree of learners' ownership of the software whose behavior is visualized by the system. Again, the levels are taken from the 2DET taxonomy. In this column, I have generally only listed the highest degree of ownership that the system allows, as systems that support learner-defined content also support teacher-defined content. However, given content is separately listed along with own content in cases where the system has a distinct mode of use specifically meant for ready-made example programs.
Notional machine elements	What the visualization covers: a non-exhaustive list of key elements of the notional machine visualized by the system. *Control* refers to control flow: the system makes explicit which part of program code is active at which stage. *Vars* stands for variables. *ExprEv* means expression evaluation. *Calls* refers to the sequencing of function/procedure/method calls and returns (which may or may not be expressed in terms of call stacks). *Refs* stands for references and/or pointers, *Addrs* for memory addresses. *Objs* is short for objects. *Classes* means that the system visualizes classes not only as source code or a static class diagram, but as a part of the program runtime. *Structs* refers generally to any composite data – arrays, records, lists, trees, and the like – that has a bespoke representation within the system. All these categories are my own abstractions, which are realized in different ways in different systems. It has not been possible to try all the systems; the descriptions given are an interpretation of the descriptions and images in the literature. All the systems reviewed depict control flow in one way or another; this is not listed separately.
Representation	What the visualization primarily consists of, on the surface: the kinds of visual elements used.
Step grain	The size of the smallest step with which the user can step through the program. *Statement* means that an entire statement, definition or declaration (usually a single line) is executed at once, although stepping into and out of functions/methods may be a separate step. *Expression* means that the user steps through stages of expression evaluation in more detail.

Table 11.5: A summary of selected program visualization systems. See Table 11.4 for a legend.

System name	Page	At least since	Status	Used for	Paradigm	Programming language	Evaluation
'regular visual debuggers'	151	varies	varies	debugging	varies	varies	varies
Basic Programming	155	1979	inactive	development	imp	BASIC	unknown
LOPLE / DynaMOD / DynaLab	156	1983	inactive	examples	imp	Pascal, [Ada], [C]	survey
Amethyst	156	1988	inactive	debugging	imp	Pascal	none
Bradman	156	1991	inactive	debugging	imp	C	experimental
EROSI	156	1996	inactive	examples	imp	Pascal	survey, qualitative
VisMod	156	1996	inactive	development, debugging, [AV]	imp	Modula-2	anecdotal
(Fernández et al.)	159	1998	inactive	examples	OO	Smalltalk	none
DISCOVER	159	1992	inactive	development	imp	pseudocode	experimental
(Kasmarik and Thurbon)	160	2000	inactive	debugging?	imp, OO	Java	experimental
CMeRun	160	2004	inactive	debugging	imp	$C{+}{+}_{ss}$	anecdotal
(Korsh and Sangwan)	160	1998	inactive	debugging	imp	$C{+}{+}_{ss}$	none
VINCE	160	1998	inactive	debugging	imp	C	experimental, survey
OGRE	160	2004	inactive?	debugging	imp, OO	C++	experimental, survey
JAVAVIS	162	2002	inactive	debugging	imp, OO	Java	anecdotal
(Seppälä)	162	2003	inactive	debugging	imp, OO	Java	none
OOP-Anim	162	2003	inactive	debugging	imp, OO	Java	none
JavaMod	162	2004	inactive	debugging	imp, OO	Java	none
JIVE	162	2002	active	debugging	imp, OO	Java	anecdotal
Memview	162	2004	inactive?	debugging	imp, OO	Java	anecdotal
JavaTool	162	2008	active	development, assignments	imp	$Java_{ss}$	none
PlanAni	164	2002	active?	examples	imp	Pascal, Java, C, Python	experimental, qualitative
Metaphor-based OO visualizer	164	2007	active?	examples	OO	Java	experimental
Eliot / Jeliot I	165	1996	inactive	debugging, AV	imp, [func]	C, $Java_{ss}$, [Scheme]	survey, qualitative
Jeliot 2000 / Jeliot 3	165	2003	active	debugging, [development], [assignments]	imp, OO	Java, [C], [Python]	experimental, survey, qualitative
GRASP / jGRASP	169	1996	active	development, debugging, AV	imp, OO	Java, C, C++, Objective-C, Ada, VHDL	experimental (on AV), anecdotal
The Teaching Machine	170	2000	active	debugging, AV, [assignments]	imp, OO	C++, Java	survey
VIP	170	2005	active	debugging	imp	$C{+}{+}_{ss}$	experimental, survey, qualitative
(Miyadera et al.)	170	2006	inactive	examples	imp	C_{ss}	none
ViLLE	173	2005	active	examples, assignments	imp	$Java_{ss}$, $C{+}{+}_{ss}$, $Python_{ss}$, PHP_{ss}, $JavaScript_{ss}$, pseudocode	experimental, survey
Jype	175	2009	inactive?	development, assignments, AV	imp, OO	Python	anecdotal
Online Python Tutor	175	2010	active	debugging, assignments	imp, OO	Python	none
WinHIPE	175	1998	inactive?	debugging, development	func	Hope	experimental, survey
CSmart	175	2011	active	assignments	imp	C	survey
(Gilligan)	177	1998	inactive	development	imp, [OO]	Pascal, [Java]	none
ViRPlay3D2	177	2008	unknown	software design, examples	OO	N/A (CRC cards)	none
(Dönmez and İnceoğlu)	179	2008	unknown	examples, assignments	imp	$C\#_{ss}$	none
Online Tutoring System	181	2010	inactive	assignments	imp	VBA_{ss}	experimental, survey
JV^2M	183	2003	unknown	assignments	bytecode	Java / bytecode	none
UUhistle	192	2009	active	assignments, debugging	imp, OO	Python, [Java]	experimental, survey, qualitative

Table 11.6: A summary of how selected program visualization systems present notional machines and engage learners. See Table 11.5 for an overview of the systems and Table 11.4 for a legend.

System name	Notional machine elements	Representation	Step grain	Direct engagement with visualization	Content ownership
'regular visual debuggers'	Control, Vars, Calls, Objs, Refs, Structs (e.g.)	standard widgets	statement	controlled viewing	own content
Basic Programming	Control, Vars, ExprEv	symbols	expression	controlled viewing	own content
LOPLE / DynaMOD / DynaLab	Control, Vars, Calls	symbols	statement	controlled viewing	own cases, [own content]
Amethyst	Control, Vars, Calls, Structs, [Refs]	abstract 2D	statement?	controlled viewing	own content
Bradman	Control, Vars, ExprEv, Refs	symbols, explanations, [abstract 2D], [smooth animation]	statement	controlled viewing	own content
EROSI	Control, Vars, Calls	abstract 2D, [audio]	statement	controlled viewing	given content
VisMod	Control, Vars, Refs, Calls, Structs	standard widgets, abstract 2D	statement	controlled viewing	own content
(Fernández et al.)	Refs, Objs, Classes, Calls	abstract 2D, visual metaphors	message passing	controlled viewing	modified content
DISCOVER	Control, Vars	abstract 2D	statement	controlled viewing	own content
(Kasmarik and Thurbon)	Control, Vars, Refs, Calls, Objs, Structs	abstract 2D	statement	controlled viewing	own content?
CMeRun	Control, Vars, ExprEv	text	statement	controlled viewing	own content
(Korsh and Sangwan)	Control, Vars, Refs, Calls, ExprEv, Structs	abstract 2D	expression	controlled viewing	own content
VINCE	Control, Vars, Refs, Calls, Addrs, Structs	abstract 2D, explanations	statement	controlled viewing	own content
OGRE	Control, Vars, Refs, Objs, Classes, Calls, Structs	abstract 3D, smooth animation, explanations	statement	controlled viewing	own content
JAVAVIS	Vars, Refs, Objs, Calls, Structs	abstract 2D, UML	statement	controlled viewing	own content
(Seppälä)	Control, Vars, Refs, Objs, Calls	abstract 2D	statement	controlled viewing	own content
OOP-Anim	Control, Vars, Refs, Objs, Classes	abstract 2D, smooth animation, explanations	statement	controlled viewing	own content
JavaMod	Control, Vars, ExprEv, Refs, Objs, Calls, Structs	standard widgets, UML	expression	controlled viewing	own content
JIVE	Control, Vars, Refs, Objs, Classes, Calls, Structs	standard widgets, abstract 2D	statement	controlled viewing	own content
Memview	Control, Vars, Refs, Objs, Classes, Calls, Structs	standard widgets	statement	controlled viewing	own content
JavaTool	Control, Vars, Structs	abstract 2D	statement	controlled viewing	own content
PlanAni	Control, Vars, ExprEv, Structs	visual metaphors, smooth animation, explanations	expression	controlled viewing	own cases
Metaphor-based OO visualizer	Control, Vars, ExprEv, Refs, Objs, Classes, Calls, Structs	visual metaphors, smooth animation, explanations	expression	controlled viewing	own cases
Eliot / Jeliot I	Control, Vars, [ExprEv], Structs	abstract 2D, smooth animation	event-based	controlled viewing	own content
Jeliot 2000 / Jeliot 3	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	abstract 2D, smooth animation	expression	controlled viewing, [responding]	own content
GRASP / jGRASP	Control, Vars, Calls, Refs, Objs, Structs	standard widgets, abstract 2D, smooth animation	statement	controlled viewing	own content
The Teaching Machine	Control, Vars, ExprEv, Calls, Addrs, Refs, Objs, Structs	standard widgets, abstract 2D	expression	controlled viewing	own content
VIP	Control, Vars, ExprEv, Calls, Refs, Structs	standard widgets, explanations, [abstract 2D]	statement	controlled viewing	own content
(Miyadera et al.)	Control, Vars, ExprEv, Calls	abstract 2D	statement?	controlled viewing	given content
ViLLE	Control, Vars, Calls, Structs	standard widgets, explanations	statement	controlled viewing, responding, [applying]	given content, [modified content]
Jype	Control, Vars, Objs, Refs, Calls, Structs	standard widgets, abstract 2D	statement	controlled viewing	own content
Online Python Tutor	Control, Vars, Objs, Classes, Refs, Calls, Structs	abstract 2D	statement	controlled viewing	own content
WinHIPE	Control, Vars, ExprEv, Refs, Calls, Structs	abstract 2D	expression	controlled viewing, applying	own content / given content
CSmart	Control, Vars, ExprEv	explanations, visual metaphors, audio	statement	controlled viewing	given content (model solution)
(Gilligan)	Control, Vars, ExprEv, Calls, Structs, [Objs], [Classes], [Refs]	visual metaphors, standard widgets	expression	applying	own content
ViRPlay3D2	Vars, Objs, Classes, Refs, Calls	virtual 3D world	message passing	applying (when designing) / controlled viewing (scripted mode)	own content (when designing) / given content (scripted mode)
(Dönmez and İnceoğlu)	Control, Vars, ExprEv	standard widgets	expression	applying	own code
Online Tutoring System	Control, Vars, ExprEv	standard widgets, explanations	expression	applying	given content
JV^2M	the Java Virtual Machine	virtual 3D world	bytecode instruction	applying	given content
UUhistle	Control, Vars, ExprEv, Calls, Refs, Objs, Classes, Structs	abstract 2D, smooth animation, explanations	expression	applying, responding, controlled viewing	given content / own content

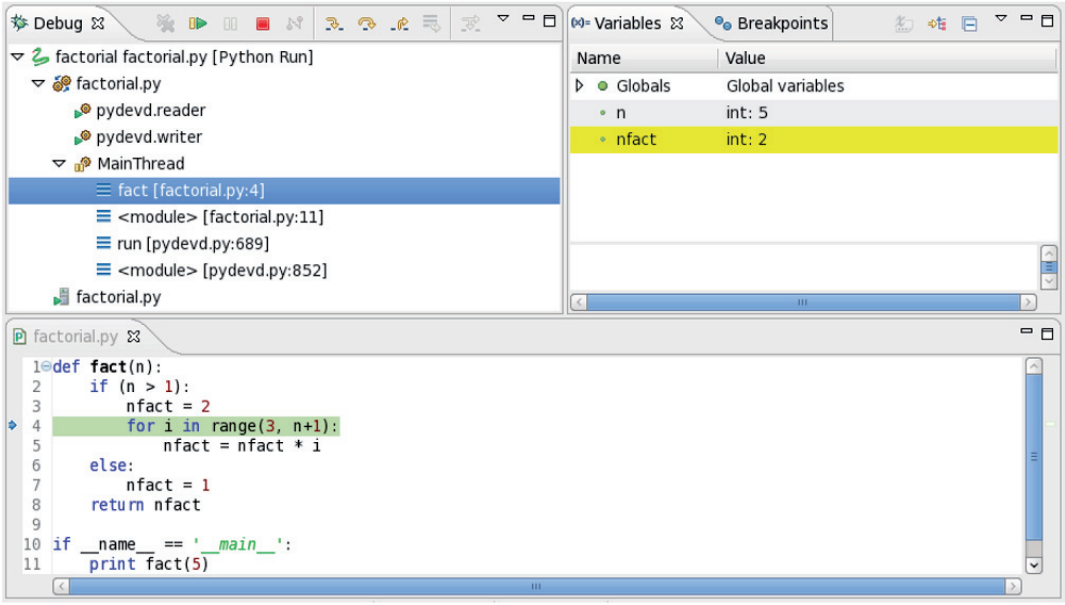


Figure 11.4: The PyDev debugger for Python programs within the Eclipse IDE (image from Helminen, 2009). A Python program is being executed, with line 4 up next. Threads and call stacks are listed in the top left-hand corner. Global and local variables are shown on the right. The yellow highlight signifies a change in the value of a variable.

programs may be large in terms of both code and data. For such reasons, the visualization shown by a typical visual debugger is not particularly graphic, and consists primarily of text within standard GUI widgets.

Bennedsen and Schulte (2010) conducted an experimental study in which a group of CS1 students used the visual debugger built into the BlueJ IDE to step through object-oriented programs, while a control group used manual tracing strategies. They found no significant differences in the performance of the groups on a post-test of multiple-choice questions on program state. A rerun of the experiment using a different debugger yielded similar results. Bennedsen and Schulte surmise that "it could be that the debugger is not useful for understanding the object interaction but just for finding errors in the program execution" (p. 18).

Despite their limitations, visual debuggers are worth a mention in this section because they are highly useful tools that novices do encounter in CS1 courses, because they do visualize certain aspects of program dynamics, and because they serve as a point of departure for reviewing the more education-oriented systems below.

11.3.2 Many educational systems seek to improve on regular debuggers

Most of the systems reviewed in this chapter are program animation tools that can be thought of as educators' attempts to improve on regular visual debuggers. Some of these systems look very similar to regular visual debuggers, others feature more unusual visualizations.

An early system on the Atari: Basic Programming

An early educational PV system that supported visual tracking of program execution was *Basic Programming*, "an instructional tool designed to teach you the fundamental steps of computer programming" (Robinett, 1979). Shown in Figure 11.5, Basic Programming was an integrated environment

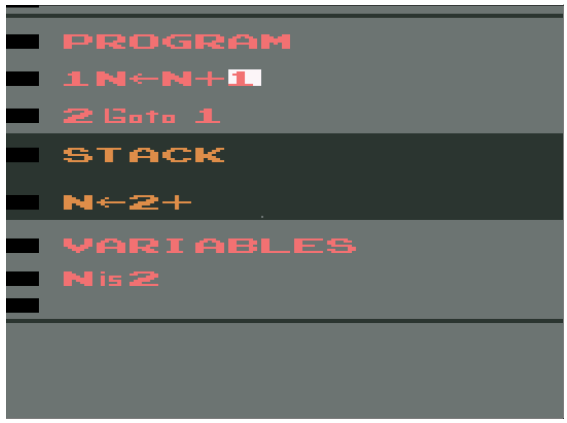


Figure 11.5: A part of the Basic Programming environment on the Atari 2600 (image from Kelleher and Pausch, 2005). A snapshot of a tiny program in mid-execution is shown. The Program panel contains BASIC code. The Variables panel shows the current values of variables. The Stack panel shows the stages of expression evaluation: here 1 is just about to be added to 2, which is the current value of `N`.

for the Atari 2600 computer, in which the user could input BASIC code and view its execution. The Stack panel of the display showed the stages of expression evaluation in more detail than a regular visual debugger does, as illustrated in Figure 11.5. The system also featured a 2D graphics area for displaying sprites.

Libraries of animated examples: LOPLE, DYNAMOD, and DynaLab

Although many software visualization systems were developed in the 1980s, few (that I am aware of) fall within the scope of this review, as education-oriented systems tended to deal with algorithm visualization (e.g., BALSA; see Brown, 1988) and program visualization systems were intended for expert use (e.g., VIPS; see Isoda et al., 1987). An exception to this trend was *LOPLE*, a dynamic Library Of Programming Language Examples (Ross, 1983). LOPLE was designed to allow novices to step through the execution of given example programs. The manner of execution in LOPLE was similar to that of a modern visual debugger.

LOPLE evolved first into *DYNAMOD* (Ross, 1991) and then into *DynaLab* (Birch et al., 1995; Boroni et al., 1996). DynaLab allowed execution to be stepped backwards, a feature that novices prized, according to student feedback. The earliest work was done with Pascal; the platform later supported various other programming languages as well. A screenshot of DynaLab is shown in Figure 11.6.

The authors provide anecdotal evidence of the usefulness of the tool in their teaching (Ross, 1991; Boroni et al., 1996). Their students liked it, too (Ross, 1991).

Towards more graphical systems: Amethyst, Bradman, EROSI, and VisMod

Myers et al. (1988) presented an early PV system prototype called *Amethyst*. Amethyst visualized data as two-dimensional graphics during a program run (Figure 11.7). The system differed from the earlier algorithm visualization systems from which it was derived in that it created visualizations automatically for any program. The user would nevertheless manually mark through the GUI which data items should be visualized.

Smith and Webb (1991, 1995a) created *Bradman*, a visual debugger intended for novice C programmers. The explicit goal of the system was to improve students' mental models of program

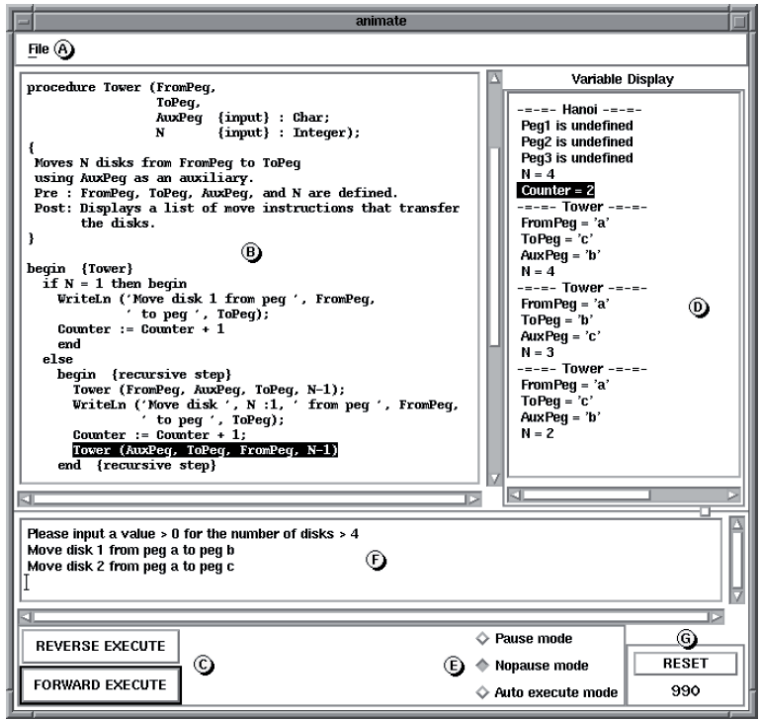


Figure 11.6: DynaLab executing a Towers of Hanoi example program (Pratt, 1995).

execution by helping them visualize the dynamic behavior of programs. As with any debugger, a user of Bradman could put in their own code and examine its behavior statement by statement. Compared to regular debuggers, a novelty in Bradman was the detailed English explanations of each statement as it was executed. Smith and Webb (1995b) report that students liked these explanations and reacted particularly positively to a version of Bradman that included them in comparison to one that did not. The other major novelty in Bradman was its illustration of changes in program state. Bradman showed previous and current states of a program side by side for convenient comparison. This is pictured in Figure 11.8, which also illustrates how Bradman visualized references using graphical arrows. The explicit treatment of state changes was particularly useful since Bradman did not support stepping backwards as, for instance, DynaLab did. Smith and Webb (2000) report on an experimental evaluation of Bradman in which they found that CS1 students who used Bradman for examining teacher-given programs performed significantly better in a multiple-choice post-test on parameter passing than did students without access to Bradman. Other, similar tests performed during the intervention did not yield statistically significant differences between groups.

George (2000a,b,c, 2002) evaluated a system called *EROSI*, which was built primarily for illustrating procedure calls and recursion. EROSI featured a selection of program examples, whose execution it displayed. Subprogram calls were shown in separate windows with arrows illustrating the flow of control between them. George demonstrated through analyses of student assignments and interviews that the tool was capable of fostering a viable 'copies' model of recursion which students could then apply to program construction tasks. The students liked it.

VisMod (Jiménez-Peris et al., 1999) was a beginner-friendly programming environment for the Modula-2 language. The system had various features designed with the novice programmer in mind, including a pedagogically oriented visual debugger (Figure 11.9). The system had a cascading-windows representation for the call stack, as well as line graphics of certain data structures (lists and trees). The authors report that students liked it.

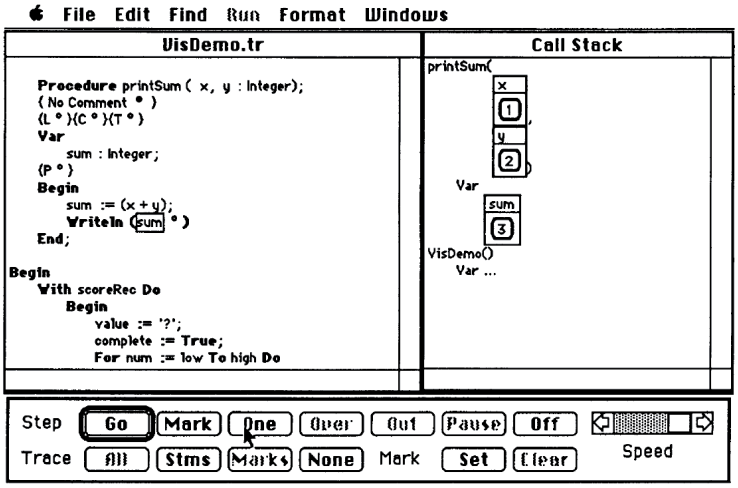


Figure 11.7: A Pascal program in Amethyst (Myers et al., 1988). Amethyst used different shapes to emphasize the different types of values; for instance, the rounded rectangles in this figure denote integer values.

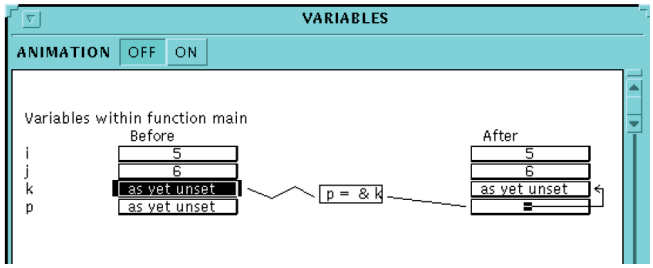


Figure 11.8: One of Bradman's GUI windows after the C statement `p = & k` has just been executed (Smith and Webb, 1995a). The previous state is shown side by side with the current state. (The rest of the program does not appear in the Variables window.)

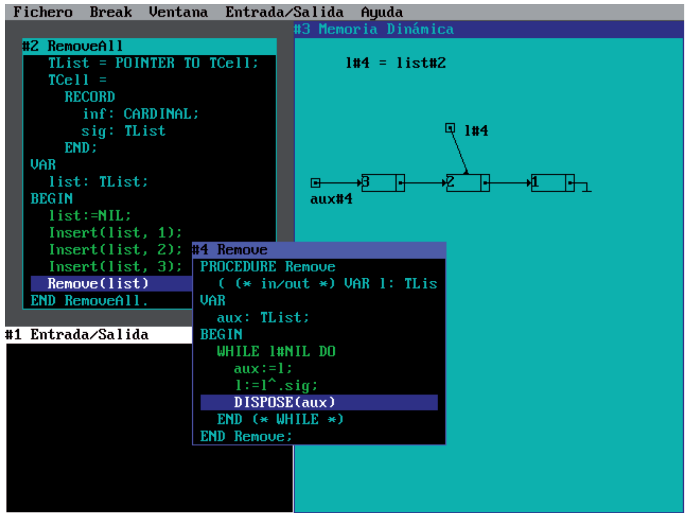


Figure 11.9: A Modula-2 program in VisMod (Jiménez-Peris et al., 1999).

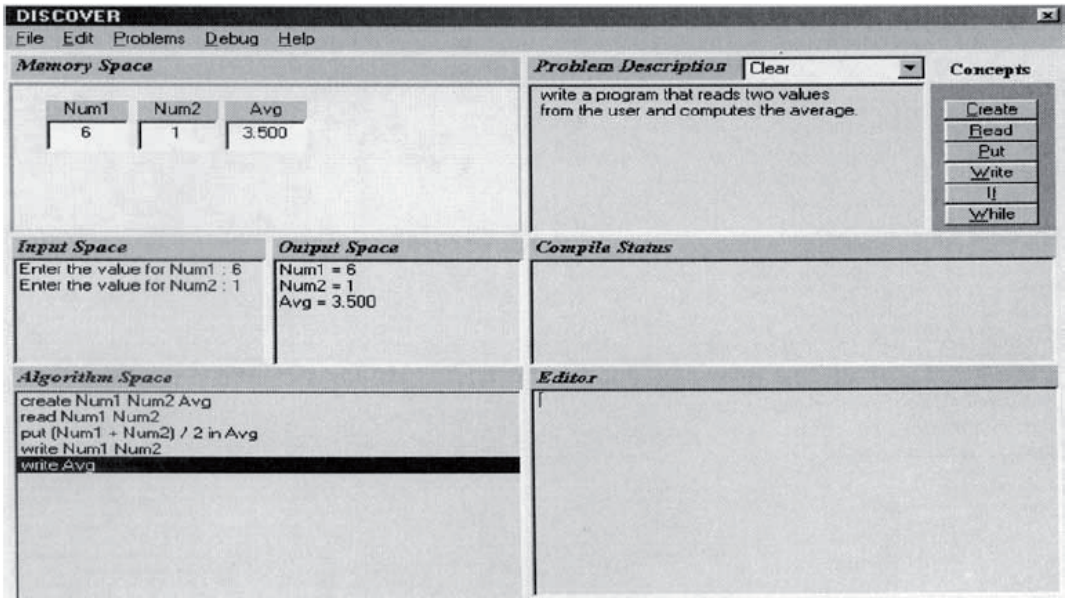


Figure 11.10: The user has just completed writing a program in DISCOVER (image from Ramadhan et al., 2001). He has used the editor to input pseudocode into the Algorithm Space. In the immediate execution mode, the effects of each new statement on execution instantly shown through updates to the other panels.

A high-level objects view: Fernández et al.'s tool

Fernández et al. (1998) presented a system for illustrating the inner behavior of Smalltalk programs. Their innominate system was built on top of a more generic learning platform called LearningWorks, which used a visual book metaphor to present information. Their tool visualized the dynamic object relationships within teacher-defined example systems of classes by drawing object diagrams in which interobject references and messages passed were shown as arrows of different kinds.

Immediate visualization: DISCOVER

DISCOVER (see Ramadhan et al., 2001, and references therein) was a prototype of an intelligent tutoring system which featured an explicit conceptual model of a notional machine in the form of a visualization (Figure 11.10). I discuss here only the DISCOVER's software visualization features.

One way of using DISCOVER was similar to that found in many of the other tools reviewed: the user could step through an existing program's execution and observe changes in variables, etc., with the help of a graphical machine model. A more unusual feature of the system was the (optional) immediacy of the visual execution of partial solutions during editing: the user could type in an additional statement into their programs upon which DISCOVER would instantly show onscreen the effects of the new statement on the entire program when executed.

Some of the evaluations of DISCOVER examined the impact of the visualization. In an experiment, students who used a visualization-enabled version of the system made fewer errors and completed tasks quicker than other students who used a version that did not visualize execution or support line-by-line stepping through execution, although the result was not statistically significant (Ramadhan et al., 2001; see also Ramadhan, 2000).

Producing snapshots: Kasmarik and Thurbon's tool and CMeRun

Kasmarik and Thurbon (2003) used an unnamed system that could produce visualizations of specific program examples. Their tool took Java code as input and produced a sequence of graphical diagrams to illustrate the given program's states, in particular the values that variables get at different stages of execution. The authors evaluated their visualizations (which had been created by the teacher using the tool) experimentally in a CS1 course. They had students answer questions about small example programs, which were additionally accompanied by visualizations in the case of the treatment group. The treatment group's results were significantly better than the control group's, without a significant increase in the time they spent on the task.

Etheredge (2004) described a tool called *CMeRun*, designed to aid novice programmers debug their programs. CMeRun instrumented C++ code in such a way that when the code was run, an execution trace was printed out. The user could examine the trace to see which statements were executed and what the values of variables were at each stage. For instance, an execution of the line

```
for (i = 0; i <= SIZE; i++)
```

might appear in the trace of a CMeRun-augmented program as

```
for (i = 0; i<3> <= SIZE<4>; i<3>++)
```

Etheredge reports a positive response from teacher and student evaluators.

Memory models in C/C++: Korsh and Sangwan's tool, VINCE, and OGRE

A prototype system for visualizing the behavior of C++ programs for CS1 and CS2 students was presented by Korsh and Sangwan (1998). Their tool used abstract graphics (mostly boxes inside boxes; see Figure 11.11) to visualize the values of variables, the call stack, and the stages of expression evaluation. The system required the user to annotate the source code to indicate which parts he wished to visualize, which may have made it quite challenging for novices to use effectively. The user could also adjust the level of detail by selecting which operations were animated and which were not.

VINCE was a tool for exploring the statement-by-statement execution of C programs – self-written by students or chosen from a selection of given examples (Rowe and Thorburn, 2000). VINCE visualized computer memory on a relatively low level of abstraction, as a grid which illustrated where pointers point and references refer (Figure 11.12). Rowe and Thorburn (2000) compared the confidence and C knowledge of CS1 students who used VINCE for extra tutorials over a three-week period to those of a control group who did not do the extra tutorials. Their results suggest that VINCE had no significant impact on students' self-assessment of their programming ability, but the VINCE users did in fact learn more than the control group (as might be expected since they had extra learning activities). The students liked it.

A related later system, *OGRE*, visualized C++ programs using 3D graphics (Milne and Rowe, 2004). Each scope within a running program (e.g., a method activation) was represented by a flat plane on which small 3D figures appear to represent objects and variables. References and pointers were shown as arrows, and data flow as an animation in which a cylinder moved through a pipe between source and target (Figure 11.13). The user could step forward and backward in the execution, and use 3D-game-like controls for moving about, rotating the view, and zooming in and out. Milne and Rowe (2004) conducted an experimental study to determine the effectiveness of the OGRE approach. Their target group was not a CS1 course but second-year students who had just completed a course on object-oriented C++ programming. Milne and Rowe report that students who were given additional OGRE-based tutorials on certain difficult topics to complement other learning materials could answer questions on those same topics significantly better than other students who had not had that access. An additional interview study showed that their students liked OGRE, as did the instructors who had used it.

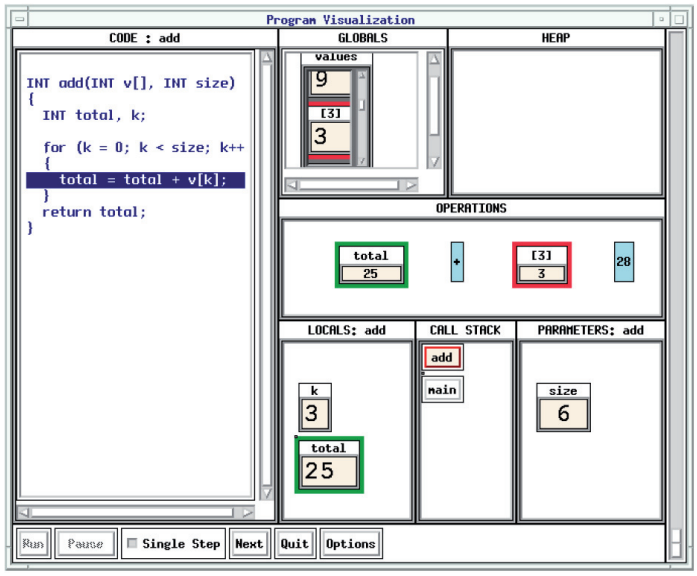


Figure 11.11: Korsh and Sangwan's (1998) program visualization tool. The user has used uppercase type declarations to mark which parts of the program should be visualized on the right. Expression evaluation is shown step by step in the Operations panel.

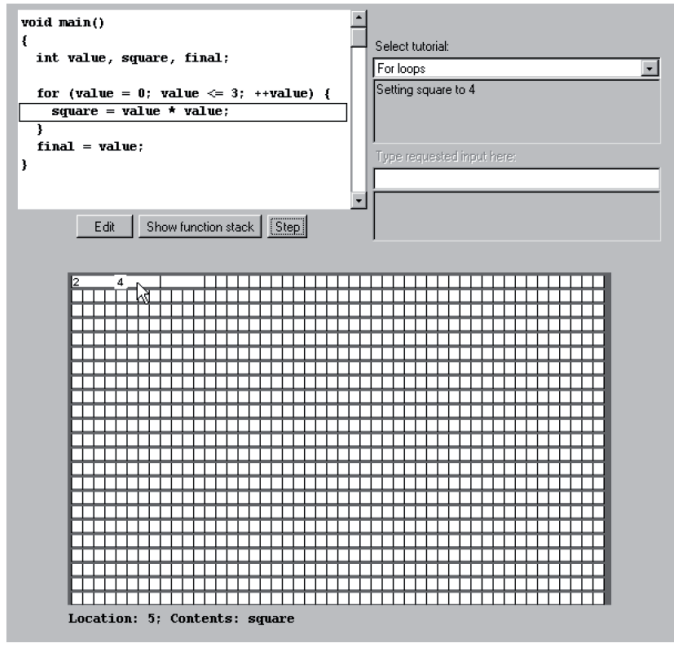


Figure 11.12: VINCE executing a C program (Rowe and Thorburn, 2000). Each square in the grid corresponds to a single byte of memory. Note that the mouse cursor is over a four-byte integer, which is currently stored in the variable `square` as indicated by the text at the bottom.

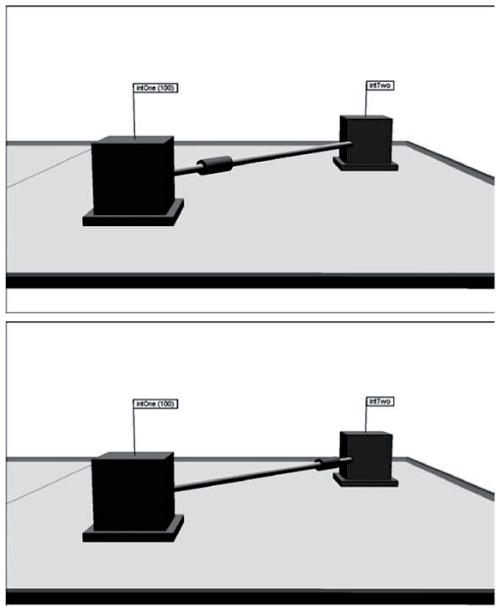


Figure 11.13: Two snapshots of the animation that OGRE used to show a value being assigned from a variable to another (Milne and Rowe, 2004).

Java animators: JAVAVIS, Seppälä's tool, OOP-Anim, JavaMod, JIVE, Memview, and JavaTool

Three distinct system prototypes from around the same time built on UML to illustrate the execution of Java programs. *JAVAVIS* (Oechsle and Schmitt, 2002) was an educationally motivated debugger, which used UML object diagrams and sequence diagrams to expose program behavior in a learner-friendly way. Unlike many of the other systems reviewed, JAVAVIS featured limited support for multithreading. Seppälä (2004) presented a similar visual debugger for CS1 use. His prototype system visualized the execution of object-oriented Java programs as dynamic object state diagrams, a notation that "attempts to show most of the runtime state of the program in a single diagram". In particular, the system visualized both method invocations and references between objects in the same graph. Seppälä's visualization essentially combined elements of the object diagrams and collaboration diagrams of UML. *OOP-Anim* (Esteves and Mendes, 2003, 2004) was a program visualization tool for object-oriented programs, which produced a step-by-step visualization of its execution, showing classes in UML, and objects as lists of instance variables and methods. Variables were visualized as small boxes which could store references to objects (shown as connecting lines between variable and object).

Gallego-Carrillo et al. (2004) presented *JavaMod*, a visual debugger for Java programs with applications in education. The primary difference between JavaMod and a regular visual debugger lies in how JavaMod treated each structural element of the code separately rather than stepping through the program line by line. For instance, the initialization, termination check, and incrementation of a `for` loop were each highlighted as separate steps in execution (cf., e.g., the Basic Programming system, above, and Jeliot and the Teaching Machine, below).

JIVE is a sophisticated debugging environment for Java programs (see, e.g., Gestwicki and Jayaraman, 2005; Lessa et al., n.d.), currently implemented as a plugin for the Eclipse IDE. JIVE is not meant exclusively for pedagogical use and its expandable visualizations (Figure 11.14) can be used also for examining larger object-oriented programs, including multithreaded ones. JIVE supports reverse stepping and has various other features beyond the scope of my review. The authors have used the system in a variety of courses, introductory-level programming among them.

The creation of the *Memview* debugger (Gries et al., 2005) was motivated by the desire to support the use of Gries and Gries's (2002; and see Section 10.3 above) teachable memory model by introducing

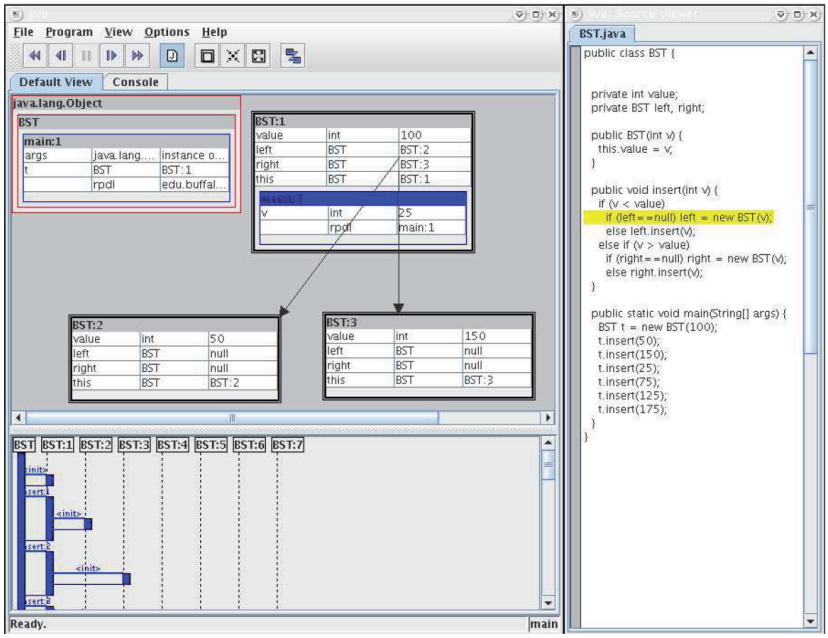


Figure 11.14: JIVE displays various aspects of a Java program's execution (Gestwicki and Jayaraman, 2005). Object relationships are illustrated on the left, with a sequence diagram of history information below it.

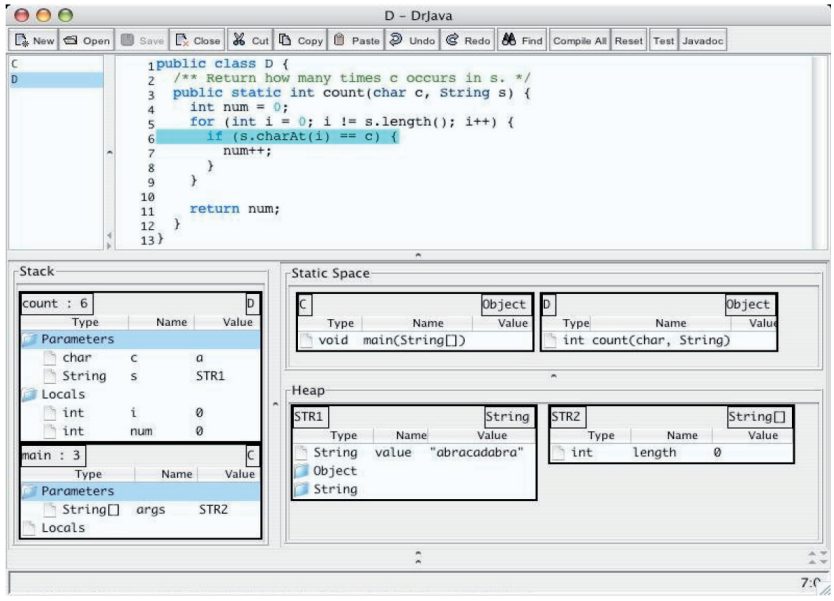


Figure 11.15: A Java program running in Memview (Gries et al., 2005). The classes `C` and `D` appear within Static Space, objects within the Heap, and frames on the Stack.

a system that automates the creation of memory diagrams. Memview, which is an add-on for the DrJava IDE (Allen et al., 2002) works much like a regular visual debugger, but has a more sophisticated and beginner-friendly way of displaying the contents of memory that works well for small CS1 programs (Figure 11.15). Gries et al. (2005) report anecdotal evidence of the tool's success in teaching.

One more tool for visualizing Java programs to novices is *JavaTool* (Pereira Mota et al., 2009; Rossy de Brito et al., 2011). The system is similar to a regular visual debugger, but features beginner-friendly controls for stepping through an animation of the program. Only a small subset of Java is supported (no objects; only a handful of standard functions can be called). JavaTool's most distinguishing feature is that it is designed as a plugin for the popular Moodle courseware. It is intended to be used for small programming assignments in which the student writes and debugs code in their web browser and submits it for grading, receiving instant feedback from the system and optionally peer feedback from other students.

Roles and metaphors: PlanAni and the metaphor-based OO visualizer

Roles of variables (briefly discussed above on p. 42) are stereotypical variable usage patterns. For instance, a variable with the role 'stepper' is assigned values according to some predefined sequence, e.g., 0, 1, 2, etc., while a 'fixed value' is a variable whose value is never changed once it is initially assigned to.

Sajaniemi and his colleagues built a system called *PlanAni* for the visualization of short, imperative CS1 program examples in terms of the variables involved and in particular their roles (Sajaniemi and Kuittinen, 2003). Each variable is displayed using a role-specific visual metaphor (see Figure 11.16). For instance, "a fixed value is depicted by a stone giving the impression of a value that is not easy to change, and [...] A stepper is depicted by footprints and shows the current value and some of the values the variable has had or may have in the future, together with an arrow giving the current direction of stepping" (Sajaniemi and Kuittinen, 2003, p. 11). PlanAni visualizes operations on these variables (e.g., assignment) as animations which, again, are role-specific.

PlanAni cannot visualize arbitrary programs, only examples that a teacher has configured in advance (in Pascal, C, Python, or Java). The teacher can include explanations to be shown at specific points during the execution sequence. Through roles, the system aims to develop students' plan schemas (see Section 4.4.1).

Sajaniemi and Kuittinen (2003) report that PlanAni had a positive impact on in-class discussions compared to a control group that used a regular visual debugger. The students liked it, and worked for longer with PlanAni whereas the debugger group tended to surf the web more during labs. Sajaniemi and Kuittinen (2005) argue, on the basis of an experiment, that using PlanAni helped foster the adoption of role knowledge and PlanAni users understood programs more deeply than non-users did, although the deeper understanding of the PlanAni users was not significantly reflected in the correctness of their answers. Expanding upon these results, Byckling and Sajaniemi (2005) report that students using PlanAni outperformed other students in code-writing tasks and exhibited significantly more forward-developing behavior while coding, which is suggestive of increased schema use (see Section 4.4.2).

Nevalainen and Sajaniemi (2005) used eye-tracking technology to compare the targeting of visual attention of PlanAni users on the one hand and visual debugger users on the other. As might be expected, PlanAni users focused a great deal more on variables. Nevalainen and Sajaniemi also studied program summaries written by the two groups immediately after using the tool, and conclude that PlanAni increased the use of higher-level information at the expense of low-level code-related information. Nevalainen and Sajaniemi further report that students found PlanAni to be clearer but relatively unpleasant to use (because too slow) compared to a visual debugger. Nevalainen and Sajaniemi (2006) similarly compared how novice programmers used a regular PlanAni and a variant with no animations. They found that irrespective of the version of the tool, the users mainly relied on textual cues (popup windows and program code). Nevalainen and Sajaniemi conclude that the location and size of visualizations is more important than animation, and that using the role images is more significant than animating them. In yet another experiment, however, Nevalainen and Sajaniemi (2008) did not find the presence of role images to be crucial to the formation of role knowledge compared to a version of PlanAni in which only textual explanations of roles were present. Stützle and Sajaniemi (2005) found that the role metaphors used in PlanAni worked better than a neutral

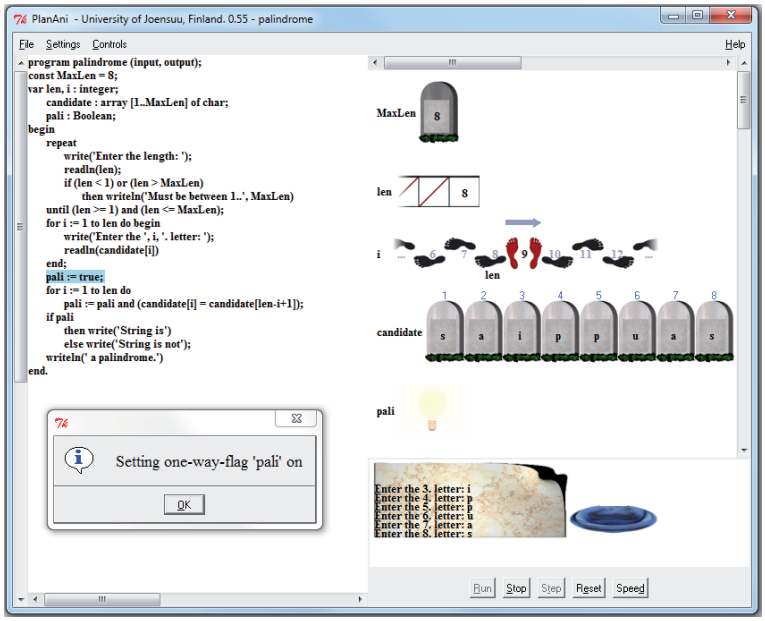


Figure 11.16: PlanAni executing a Pascal program. Variables are visualized in role-specific ways. A 'fixed value' is represented by carving a value in stone. A 'most-recent holder' is represented by a box with previous values crossed out. A 'stepper' is represented by footprints leading along a sequence of values. The lamp representing a 'one-way flag' is being switched on, as explained by the popup dialog.

set of control metaphors.

A 'sequel' to PlanAni is the metaphor-based animator for object-oriented programs envisioned by Sajaniemi et al. (2007), who recommend that their system be used by students who have first grasped some fundamental programming concepts using PlanAni and are now learning about object-oriented concepts. Their OO animator (Figure 11.17) also uses visual metaphors for variable roles but further adds support for classes ('blueprints'), objects ('filled-in pages'), references ('flags'), method calls and activations ('envelopes' and 'workshops'), and garbage collection (by a 'garbage truck'). The system is otherwise fairly similar to PlanAni. It does not work with arbitrary programs but only with predefined examples – the existing incarnation of the system is a web page on which a number of examples can be accessed as Flash animations.

Long-term tool development: from Eliot to Jeliot 3

One of the longest-lasting and most-studied program visualization tools for CS1 is *Jeliot*. Its longevity, as with almost any successful piece of software, is based on shedding its skin a few times, sometimes accompanied by a change of viscera. The stages of the Jeliot project have recently been reviewed by Ben-Ari et al. (2011).

Jeliot started out as *Eliot* (Sutinen et al., 1997), a software visualization tool that graphically animated data (variables) in user-defined C programs. In Eliot, the user selected which variables were animated, and also had a partial say in what the visualization of a program run looked like through the choice of colors and locations for the boxes that represented variables. Eliot's goals were primarily on the algorithm visualization side, and it reportedly worked best when used by programmers who had at least a bit of experience rather than by complete beginners.

Jeliot I, a Java implementation of Eliot (Haajanen et al., 1997), was a proof-of-concept system for

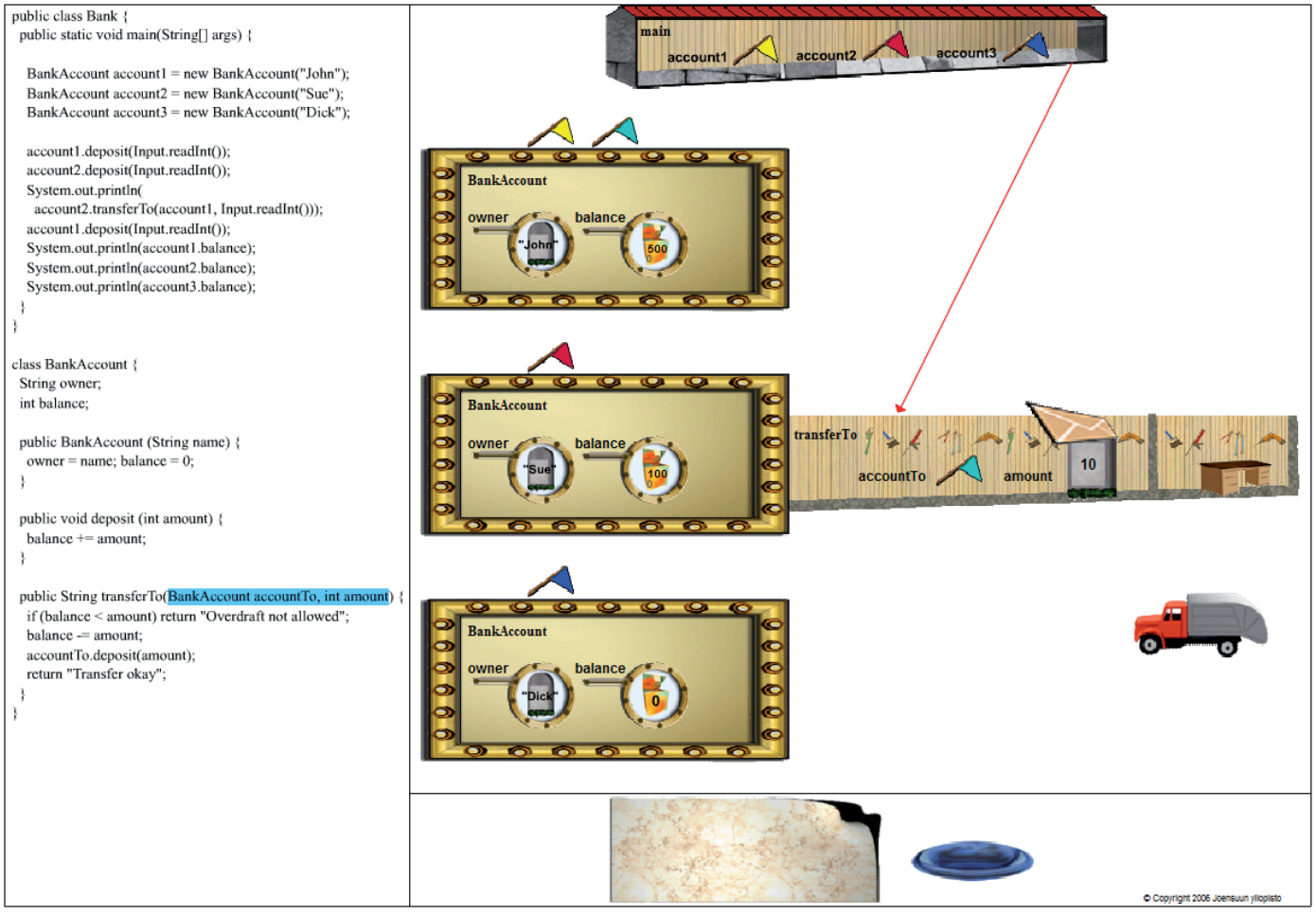


Figure 11.17: The metaphor-based OO animator of Sajaniemi et al. running a Java program. Objects are shown as filled-in pages of a blueprint book. Method invocations (here, of `main` and `transferTo`) are shown as workshops which contain local variables. References are represented by flags whose color matches a flag on the target object. The garbage truck occasionally gathers unreferenced objects.

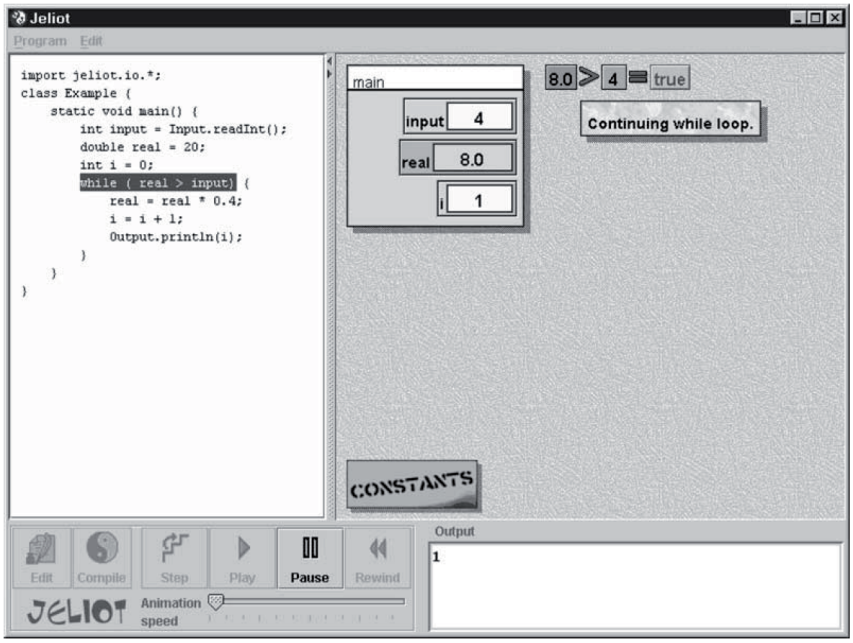


Figure 11.18: A snapshot of a Java program being executed in Jeliot 2000 (Ben-Bassat Levy et al., 2003). The code is shown on the left. On its right is shown the topmost frame of the call stack, with local variables. A conditional has just been evaluated in the top right-hand corner.

bringing program animations to the World Wide Web. Lattu et al. (2000) evaluated Jeliot I in two introductory programming courses. Some of their interviewees found Jeliot I to be helpful and suitable for beginners. Although a statistical comparison was not possible, Lattu et al. observed that students in a course whose teaching was reworked to include Jeliot I as much as possible in lectures and assignments gained much more from the experience than did students in a course that introduced the system only briefly and gave it to students as a voluntary learning aid. Lattu et al. conclude that using Jeliot I forces teachers to rethink how they teach programming. Despite the positive experiences, Lattu et al. (2000, 2003) also found problems with bringing Jeliot I to a CS1 context: the GUI was too complex for some novices to use, and many aspects of program execution, which novices would have found helpful to see, were left unvisualized (e.g., objects and classes).

Jeliot 2000 (Ben-Bassat Levy et al., 2003) was a reinvention of Jeliot as a more beginner-friendly tool, with complete automation and a more straightforward GUI (Figure 11.18). In Jeliot 2000, the user did not control the appearance of the visualization or what aspects of execution were animated. Instead, Jeliot 2000 automatically visualized Java program execution in a detailed and consistent manner, all the way down to the level of expression evaluation. Control decisions were shown as explanatory text (see Figure 11.18). The user stepped through the animation using control buttons reminiscent of a household remote control. Jeliot 2000 did not support object-oriented concepts in general, although references to array objects were graphically displayed as arrows.

Ben-Bassat Levy et al. (2003) studied introductory programming students using Jeliot 2000 in a year-long course. They found that students using the system significantly improved the results of the students who used it, with an apparent 'middle effect':

> *Even in long term use, animation does not improve the performance of all students: the better students do not really need it, and the weakest students are overwhelmed by the tool. But for many, many students, the concrete model offered by the animation can make the difference between success and failure. Animation does not seem to harm the grades neither*

> *of the stronger students who enjoy playing with it but do not use it, nor of weaker students for whom the animation is a burden.* [...] *The consistent improvement in the average scores of the mediocre students confirms the folk-wisdom that they are the most to benefit from visualization.* (p. 10)

As the improvement in grades occurred only some way into the course, Ben-Bassat Levy et al. conclude that "animation must be a long-term part of a course, so that students can learn the tool itself". Ben-Bassat Levy et al. further found that the students who used Jeliot 2000 developed a different and better vocabulary for explaining programming concepts such as assignment than did a control group that did not use Jeliot. This, the authors remind us, is particularly significant from the socio-linguistic point of view, according to which verbalization is key to understanding.

Jeliot 2000 was improved upon by adding support for objects and classes to produce *Jeliot 3* (Moreno and Myller, 2003; Moreno et al., 2004). Jeliot 3, shown in Figure 11.19, can be used as a standalone application or as a plugin for the pedagogical IDE BlueJ (Myller et al., 2007a). Kirby et al. (2010) created a variation of Jeliot for visualizing C programs by combining the user interface of Jeliot 3 with a C++ interpreter taken from the VIP tool (see below). Pears and Rogalli (2011) extended Jeliot to generate prediction questions that students can answer with their mobile phones in classroom situations.

A number of studies have investigated Jeliot 3 in practice.

Kannusmäki et al. (2004) studied the use of Jeliot 3 in a distance education CS1 course, in which using the tool was voluntary. They report that their weakest students in particular found Jeliot helpful for learning about control flow and OOP concepts, but some other students chose not to use the tool at all.

Moreno and Joy (2007) and Sivula (2005) conducted small-scale studies of students using Jeliot 3 in first-year programming courses. All of Moreno and Joy's students – who were volunteers – found Jeliot 3 easy to use, and most of those who tried it continued to use it voluntarily for debugging. Despite this, Moreno and Joy found that their students did not always understand Jeliot's animations and failed to apply what they did understand. Sivula also reports positive effects of Jeliot on motivation and program understanding, but points to how the students he observed did not use Jeliot's controls effectively for studying programs and ignored much of what might have been useful in the visualization.

Ebel and Ben-Ari (2006) used Jeliot 3 in a high-school course, and report that the tool brought about a dramatic decrease in unwanted pupil behavior. Ebel and Ben-Ari's results came from studying a class whose pupils suffered from "a variety of emotional difficulties and learning disabilities", but had normal cognitive capabilities. Although it is unclear how well this result can be generalized to other contexts, the study does indicate that program visualization can help students focus on what is to be learned.

Bednarik et al. (2006) used eye-tracking technology to compare the behavior of novices and experts who used Jeliot 3 to read and comprehend short Java programs. They found that experts tested their hypotheses against Jeliot's animation, using Jeliot as an additional source of information. Novices, on the other hand, *relied* on the visualization, interacting with the GUI and replaying the animation more. They did not read the code before animating it.

Sajaniemi et al. (2008) studied the development of student-drawn visualizations of program state during a CS1. Parts of their findings concern the effect of visualization tools on these drawings. Some of the students used Jeliot 3 for part of the course, then switched to the metaphorical OO animations (see above), while the others did the reverse. Irrespective of the group, the great majority of students did not use visual elements that were clearly taken from the PV tools. When they did, students' visualizations appeared to be mostly influenced by whichever tool they had used most recently. Sajaniemi et al. discuss the differences in some detail, pointing out, for instance, that Jeliot users tended to stress expression evaluation more, but made more errors when depicting objects and their methods.

Myller et al. (2009) investigated the use of Jeliot 3 in a collaborative learning setting. CS1 students worked in small groups in a number of laboratory sessions, during which their behavior was observed by the researchers. They found that students were especially interactive when they were required to engage with Jeliot 3 by entering input to the program. Viewing the visualization made students particularly uninteractive, even when compared to moments where they were not viewing a visualization of program execution at all. Myller et al. conclude that having students merely view a visualization is not a good idea as it reduces collaboration.

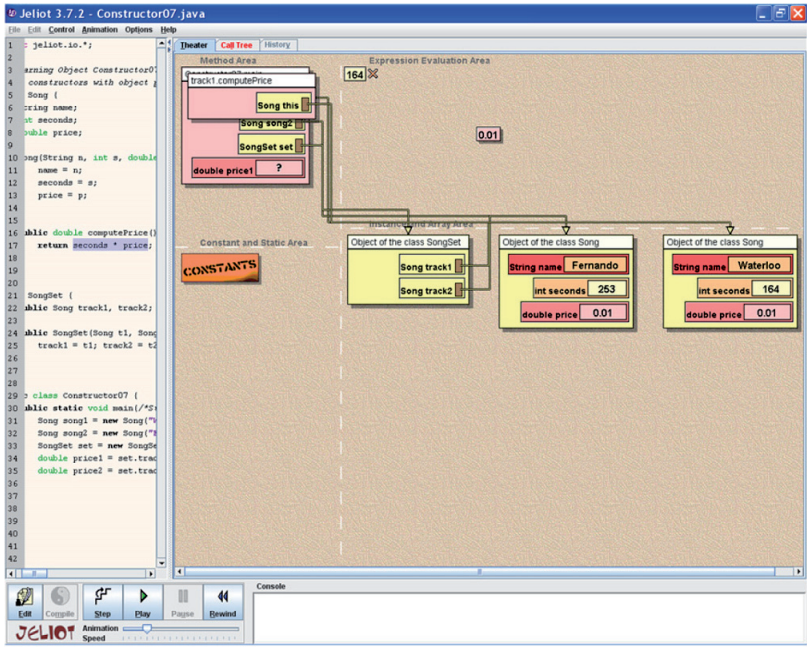


Figure 11.19: Jeliot 3 executing a Java program. The visualization extends that of Jeliot 2000 (Figure 11.18). The main novelty of Jeliot 3 is its support for object-orientation. This picture, from Ben-Ari et al. (2011), also shows how active method calls are stacked on top of each other in the Method Area.

Ma et al. (2009, 2011) observed that Jeliot 3 helped many of their students to learn about conditionals, loops, scope, and parameter passing, but that they found its visualization of references to be too complex. The authors contend that a simpler visualization system tailored for the specific topic of object assignment seemed to be more suitable for the task.

Maravić Čisar et al. (2010, 2011) used experimental studies in two consecutive years to evaluate the impact of Jeliot 3 on a programming course. During the course, some students used Jeliot for programming and debugging in the place of a regular IDE (which includes a visual debugger). In code comprehension post-tests, the students who used Jeliot performed significantly better than the control groups. Student feedback was also positive.

A versatile IDE: GRASP / jGRASP

GRASP (Cross et al., 1996), later *jGRASP* (Cross et al., n.d.; Cross et al., 2011), is another long-lived software visualization system. The system is an IDE which features a wide array of enhancements for the benefit of the novice programmer, such as the static highlighting of control structures in code, and conceptual visualization of data structures. Of most interest for present purposes is the aspect that falls within the domain of program animation: jGRASP's visual debugger features 'object viewers', which can visualize not only certain high-level data structures but also runtime entities of CS1 interest, such as arrays, objects in general, and instance variables (Cross et al., 2011).

The original GRASP was meant for Ada programming; jGRASP has full support for several languages, including Java and C++. Experimental evaluations of jGRASP have so far revolved around its AV support in courses on data structures.

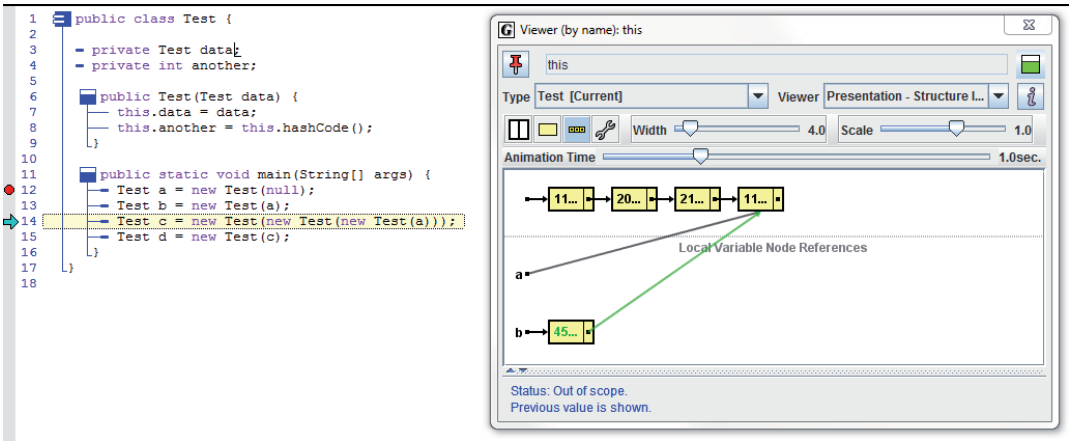


Figure 11.20: An object viewer window within the jGRASP debugger. Control has just returned from the third constructor call on line 14, and a reference is about to be assigned to variable c.

More debuggers: The Teaching Machine, VIP, and Miyadera et al.'s system

Like the later incarnations of Jeliot, *The Teaching Machine* (Bruce-Lockhart and Norvell, 2000) is a system for visualizing the execution of user code at a detailed level. The Teaching Machine visualizes C++ and Java programs in a way that is similar to, but richer than, a regular visual debugger (Figure 11.21). In particular, the novice can choose to view the stages of expression evaluation in detail to aid comprehension. Stepping back within the execution is also supported. Moreover, as shown in Figure 11.22, The Teaching Machine is capable of automatically creating dynamic diagrams of the relationships between data. Bruce-Lockhart et al. (2009) added support for teacher-defined popup questions; their interest in such questions appears to have been primarily motivated by a wish to quiz students about algorithms on a higher level of abstraction. The Teaching Machine can be used as a plugin within the Eclipse IDE.

Bruce-Lockhart et al. (2007) and Bruce-Lockhart and Norvell (2007) used The Teaching Machine in various programming courses. They give anecdotal evidence of how the system helped them teach in class. Their students liked it, especially when the system was tightly integrated with course notes. However, while the students of more advanced courses used The Teaching Machine on their own to study programs with apparent success, CS1 students tended to leave the tool alone and only viewed it while the instructor was using it. Bruce-Lockhart and Norvell (2007) report on a modified CS1 course that did not work very well, in which "we found at the end of the course that most of the first year students had never run [The Teaching Machine] on their own and consequently didn't really understand how either the [system], or the model it was based on, worked." However, after a suitable emergency intervention – extra labs in which students had to work hands-on with the system – at the end of the course, "student response was overwhelmingly positive and the course was at least rescued".

VIP is another system for visualizing C++ programs that is akin to a visual debugger but intended for a CS1 audience (Virtanen et al., 2005). It is fairly similar to The Teaching Machine, discussed above, and also displays relationships between variables (references are shown as arrows) and highlights the details of expression evaluation. VIP does not support object-orientation nor does it allow the user to step backwards within a program. VIP has a facility for displaying teacher-given hints and instructions to the student at predefined points in ready-made example programs (Lahtinen and Ahoniemi, 2007). An example is shown in Figure 11.23.

Isohanni (née Lahtinen) and her colleagues have reported on the use of VIP in a number of recent papers.

Lahtinen et al. (2007a) describe a CS1 course in which students were given the opportunity to use VIP when doing voluntary "pre-exercises" in preparation for lab sessions. Of the students who did the pre-exercises, over a quarter chose to use VIP, but traditional pen-and-paper strategies were more popular still. Lahtinen et al. also observed that the volunteers who used VIP were less likely to drop out of the

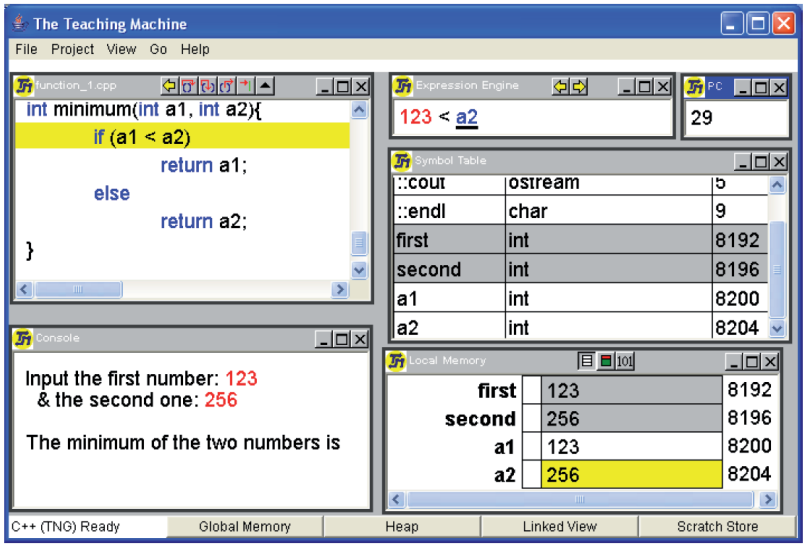


Figure 11.21: The Teaching Machine executing a C++ program (Bruce-Lockhart and Norvell, 2007). The view resembles that of a regular visual debugger, the main difference being the Expression Engine at the top, which displays expression evaluation in detail. The part of the expression that will be evaluated next is underlined.

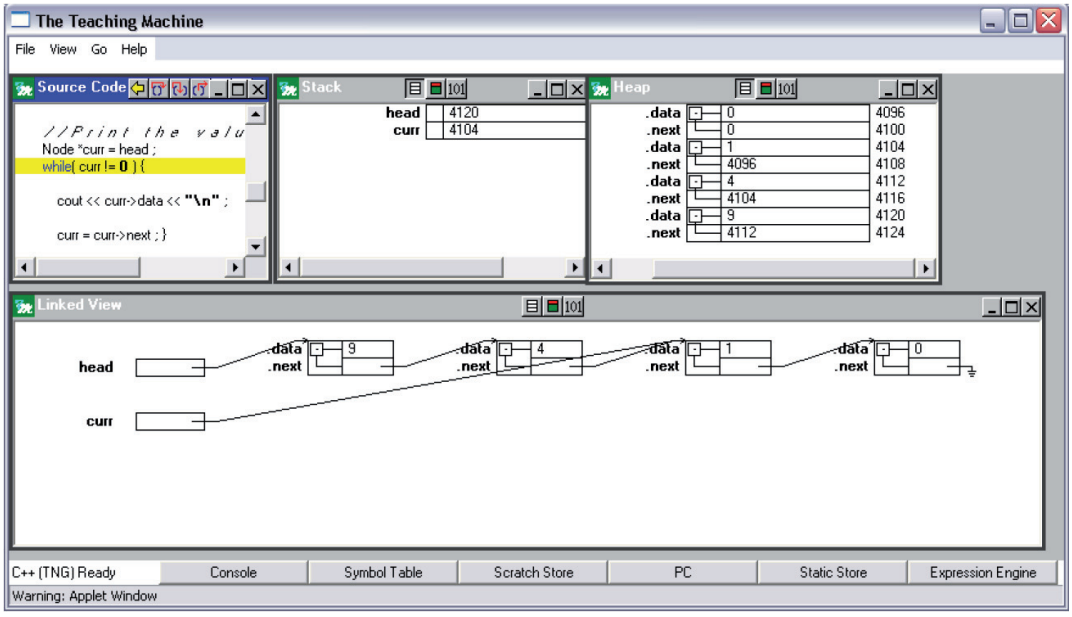


Figure 11.22: Another layout of windows within The Teaching Machine (Bruce-Lockhart and Norvell, 2007). The a stack frame and the heap are explicitly shown, along with a linked view that graphically displays relationships between data.

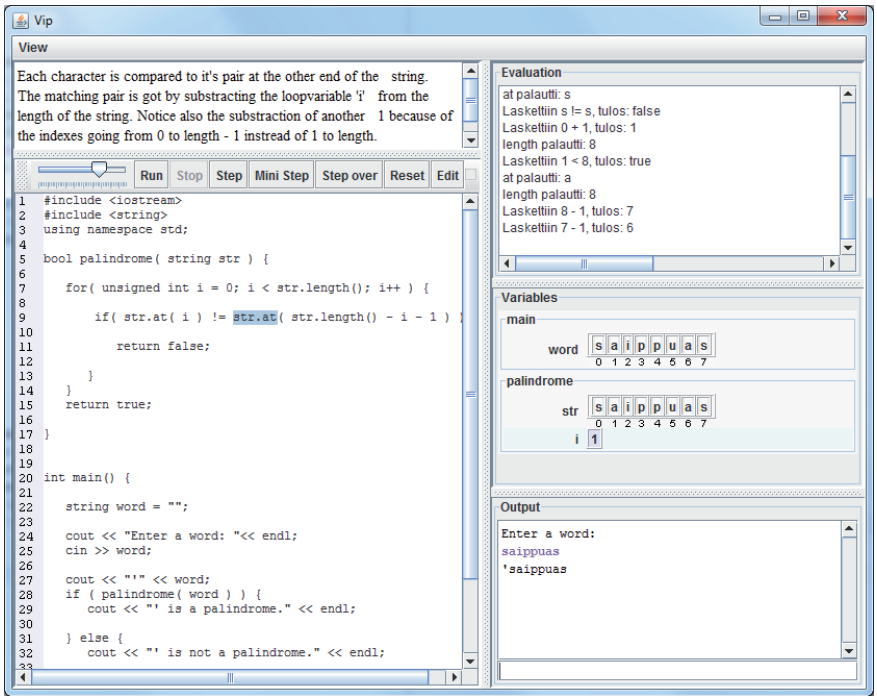


Figure 11.23: A C++ program within VIP. The `at` method of a string is just about to be called. The evaluation pane in the top right-hand corner displays previous steps; in this case, the most recent steps involve the evaluation of `at`'s parameter expression. This is a teacher-given example program, into which the teacher has embedded natural language information about individual lines of code. One such explanatory message is shown in the top left-hand corner.

course; this does not imply that VIP was responsible for this trend, however.

Lahtinen et al. (2007b) surveyed programming students in a number of European universities on their opinions of program visualization. This study was not specific to VIP, but roughly half of the students surveyed had taken a course in which VIP was used. The survey results suggest that the students who found programming challenging but manageable were the most positive about using visualizations, while the strongest and weakest students were less impressed. These findings are in line with the study of Ben-Bassat Levy et al. (2003) on Jeliot 2000, discussed above, in which a 'middle effect' was observed.

Ahoniemi and Lahtinen (2007) conducted an experiment in which randomly selected students used VIP during CS1, while a control group did not. They tested the students on small code-writing tasks. No significant differences were found when the entire treatment group and control group were considered. However, Ahoniemi and Lahtinen also identified among their students "novices and strugglers" who either had no prior programming experience or for whom the course was difficult. Of the novices and strugglers, the ones who used VIP did significantly better than the ones in the control group. Ahoniemi and Lahtinen also surveyed the students to find out how much time they had used to review materials and found that the novices and strugglers in the VIP group used more time than the ones in the control group did; this is not very surprising, since the students who used VIP had some extra materials (the visualizations and their usage instructions) to study. The authors conclude that the benefit of visualizations may not be directly due to the visualization itself but to how the visualization makes studying more interesting and leads to increased time on task.

Isohanni and Knobelsdorf (2010) qualitatively explored how CS1 students used VIP on their own. They report that students use VIP for three purposes: exploring code, testing, and debugging. Isohanni and Knobelsdorf discuss examples of ways in which students use VIP for debugging in particular. Some

students used VIP in the way intended by the teacher, that is, to step through program execution in order to find the cause of a bug. Other working patterns were also found, however. Some students stopped running the program in VIP as soon as a bug was found and from then on relied on static information visible in VIP's GUI (an extremely inefficient way of making use of VIP, the authors argue). Others still abandoned VIP entirely after they discovered a bug. Isohanni and Knobelsdorf's study shows that students do not necessarily use visualization tools in the way teachers intend them to, and underlines the need for explicit teaching about how to make use of a PV tool. Isohanni and Knobelsdorf (2011) further describe the ways in which students use VIP; their work provides an illustration of how increasing familiarity with a visualization tool over time can lead to increasingly productive and creative ways of using it.

A different system for animating C programs – with some of the same functionality as VIP and the Teaching Machine – has been created by Miyadera et al. (2007); their main research focus has been the monitoring learners' use of controls and assessment of the difficulty of different lines in code.

Variety in assignments: ViLLE

ViLLE is an online learning platform which started out as a program visualization system for CS1 that displays predefined program examples to students statement by statement (Rajala et al., 2007). My review focuses on ViLLE's program visualization functionality.

ViLLE comes with a selection of example programs; teachers can also define and share their own. The user can choose to view a program within ViLLE as either Java, C++, Python, PHP, JavaScript, or pseudocode, and change between languages at will. ViLLE supports a limited 'intersection' of these languages.

ViLLE's program visualizer has several beginner-friendly features beyond what a regular visual debugger offers, some of which are shown in Figure 11.24. The user can step forward or backwards. Each code line is accompanied by an automatically generated explanation. Arrays are graphically visualized. Teacher-defined popup questions can be embedded into programs to appear at predefined points of the execution sequence (see Figure 11.25 on page 174). ViLLE can grade the answers to multiple-choice questions in the popup dialogs and communicate with a course management system to keep track of students' scores (Kaila et al., 2008).

Apart from multiple-choice questions, ViLLE also supports a few other assignment types (Rajala et al., n.d.). Students may be required to sort lines of code that have been shuffled or to fill in some Java code to complete a program. Recent versions of ViLLE feature a "Clouds & Boxes" assignment type, which I will discuss separately in Section 11.3.3 below.

ViLLE's authors have investigated the impact of the tool with a series of experimental studies.

Laakso et al. (2008) found that previous experience with using ViLLE had a significant effect on how well high-school students learned from given examples, and conclude that to get the most out of a visualization tool, students need to be trained to use it effectively. Rajala et al. (2008) compared the performance of a treatment group using ViLLE and a control group on code-reading tasks. They found no significant differences in post-test scores between the groups. Kaila et al. (2009a) found that having students respond to ViLLE's popup questions during program execution had a better impact on learning than merely having students view example programs. Rajala et al. (2009) found that introductory programming students learned more when using ViLLE in pairs than when working alone, especially with regard to more challenging topics such as parameter passing. Kaila et al. (2010) studied three consecutive introductory programming course offerings in high school. The teaching in the courses was identical except for ViLLE being used throughout the course in the third offering. They report that using ViLLE significantly raised course grades.

Oh, and the students liked it (Kaila et al., 2009b).

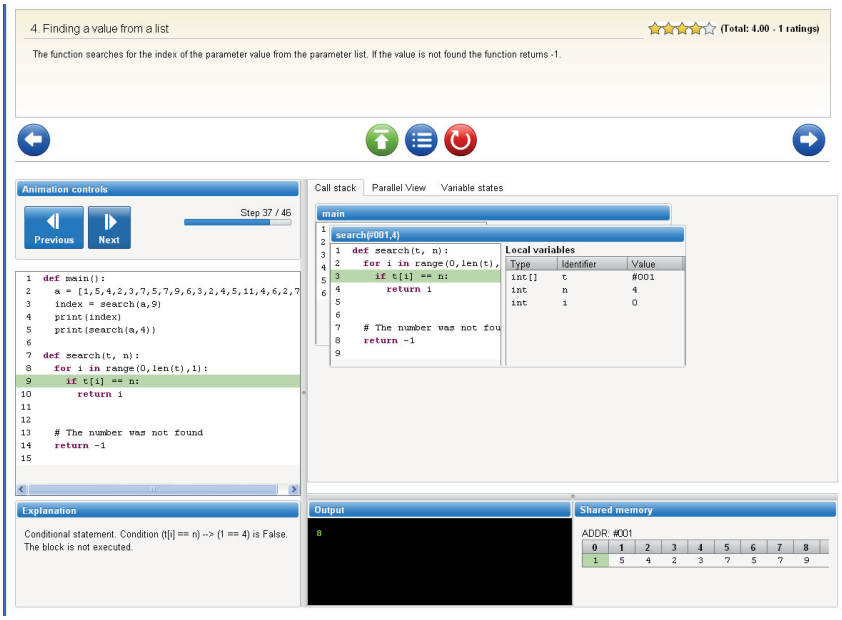


Figure 11.24: A Python program within ViLLE (image from Rajala et al., n.d.).

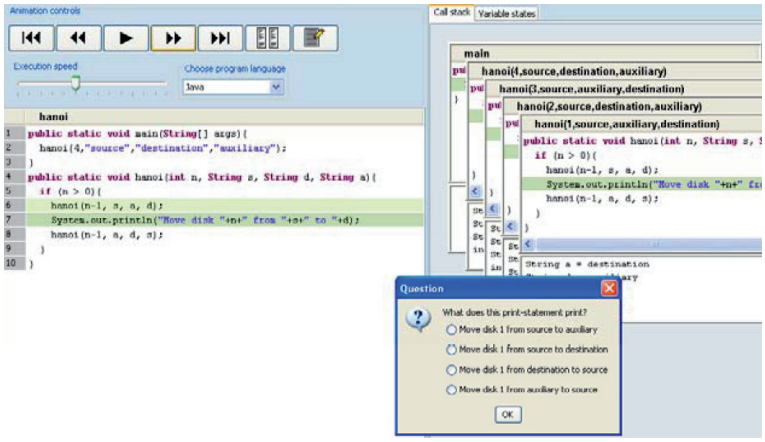


Figure 11.25: A multiple-choice question which ViLLE requires the student to answer once execution reaches a specific line of code (image from Laakso et al., 2008)

Python in the browser: Jype and the Online Python Tutor

Jype is a web-based integrated development environment for Python programming (Helminen, 2009). In addition to serving as an IDE, Jype can be used for exploring teacher-given example programs and as a platform for the automatic assessment of given programming assignments. Jype is intended specifically for CS1 use and has a number of beginner-friendly features. Among these is support for program and algorithm visualization.

Jype uses the Matrix framework (Korhonen et al., 2004) for visualizing data structures such as arrays and trees automatically when they appear in programs. Jype also visualizes the call stack in a richer way than a typical visual debugger, as illustrated in Figure 11.26.

Another new, web-based system for visualizing small Python programs is *Online Python Tutor* (Guo, n.d.), shown in Figure 11.27. Like Jype, it allows the user to step forward and backwards in a Python program at the statement level, and supports the distribution of teacher-given examples and small programming assignments with automatic feedback.

Functional programming: WinHIPE and others

WinHIPE is an education-oriented IDE for functional programming that includes a program visualization functionality (Pareja-Flores et al., 2007). WinHIPE visualizes an execution model of pure functional programs that is based on rewriting terms (no assignment). The user configures each animation by selecting which aspects are to be visualized and in what order term-rewriting proceeds in the visualization. Using this facility, teachers can produce and customize dynamic visualizations of selected examples, which can be exported for students to view. When a visualization is being viewed, the evaluation of expressions is shown step by step, and the user can step back and forth in the evaluation sequence.

Unlike many of the other visualization tools in this review, WinHIPE is designed to scale up to handle larger programs. WinHIPE is not specifically directed at CS1 – the authors have used it in more advanced courses (Pareja-Flores et al., 2007; Urquiza-Fuentes and Velázquez-Iturbide, 2007) – but is a plausible choice for beginners as well.

An earlier system for visualizing the dynamics of functional programs (in the Miranda language) was presented by Auguston and Reinfelds (1994). Mann et al. (1994) reported a study using *LISP Evaluation Modeler*, which traced the evaluation of LISP expressions. They found positive transfer effects from the use of the system for debugging, and their students were eager to use it. *ELM-ART* (see, e.g., Weber and Brusilovsky, 2001) is an intelligent LISP programming tutor for beginners, which features a component that visualizes expression evaluation. The *DrScheme*/*DrRacket* IDE for the Scheme language also uses some visual elements to present program dynamics, among its other beginner-friendly features (Findler et al., 2002).

Visualizing before writing: CSmart

CSmart (Gajraj et al., 2011) takes an approach to visualization that is quite different from all the other tools reviewed here. Instead of visualizing the execution of an existing program, the focus in CSmart is on already visualizing each statement of a C program to the student *before* the student types it in.

CSmart is an IDE with a particular pedagogical goal: to assist the student in duplicating teacher-defined example programs, thereby practicing programming fundamentals. The system knows exactly what program it requires the student to write in each assignment. It instructs the student at each step using text, graphics, and audio. Some of the guidance is teacher-annotated into the model solutions, some generated automatically. Various visual metaphors have been built into the system to illustrate the runtime semantics of the programming language. An example is shown in Figure 11.28.

Gajraj et al. (2011) report that their students liked it.

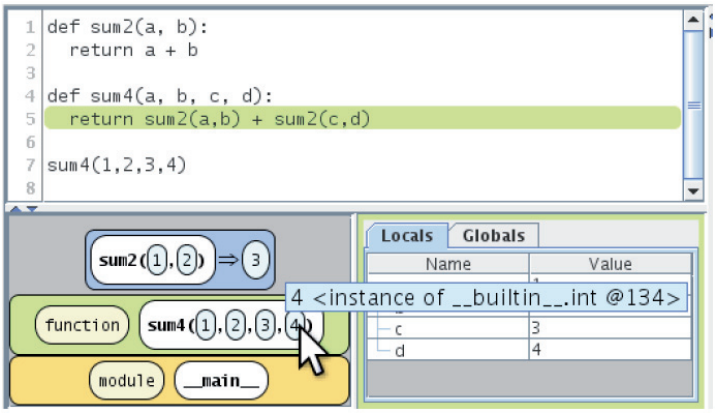


Figure 11.26: A part of Jype's user interface (Helminen, 2009). The call stack is visualized at the bottom left. The blue box signifies returning from a function.

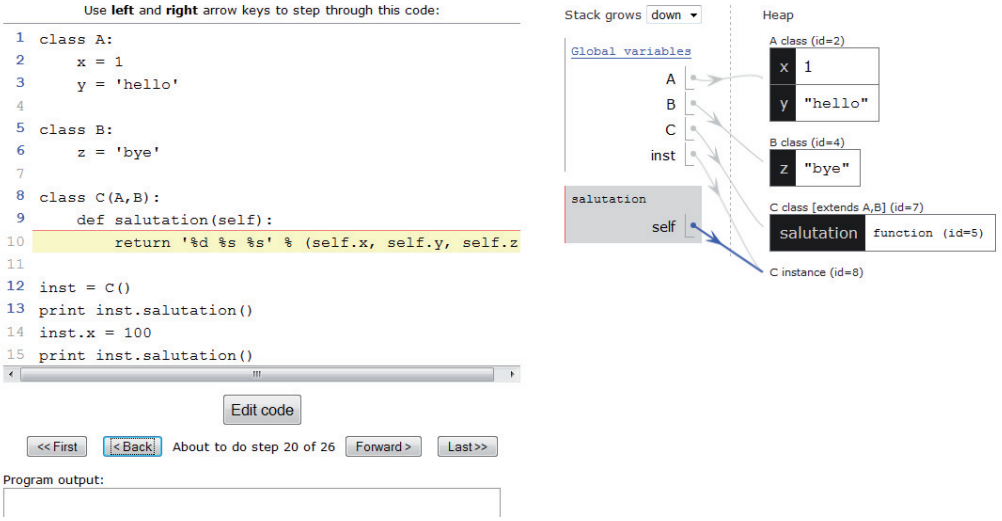


Figure 11.27: A program running in Online Python Tutor.

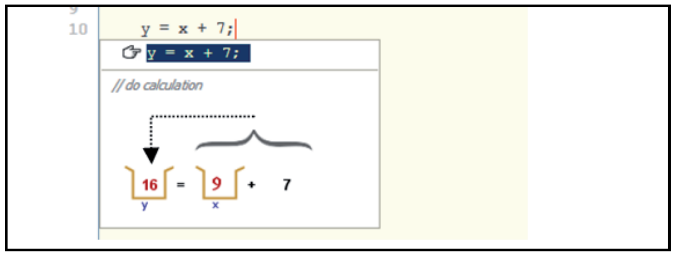


Figure 11.28: The user has just finished typing in a program within the CSmart IDE (Gajraj et al., 2011). The system's instructions to the user are still visible below line 10. In this case, they consist of the code that the student must duplicate, a short teacher-created comment, and a graphical metaphor of the statement to be written.

11.3.3 A few systems make the student do the computer's job

A few fairly recent systems have a different take on how to make use of a program visualization, and involve a high degree of interactivity. This makes them particularly relevant for this thesis. The systems discussed next are all either closely related to, or forms of, *visual program simulation* – the kind of program visualization that I define in Part IV.

Gilligan's programming-by-demonstration system

Programming by demonstration is a visual programming paradigm that uses expressive GUIs to empower end-users to program without having to write program code: "the user should be able to instruct the computer to "Watch what I do", and the computer should create the program that corresponds to the user's actions" (Cypher, 1993). The first programming-by-demonstration system was Pygmalion (Smith, 1975); many others have been created since (see, e.g., Cypher, 1993).

By and large, programming-by-demonstration systems – even those whose target audience is programming beginners – fall into the empowering systems category in Kelleher and Pausch's taxonomy (Figure 11.2) and out of the scope of this review. However, I am aware of one programming-by-demonstration system whose goals were different enough to justify an exception.

Gilligan (1998) created a prototype of a programming-by-demonstration system for novice programmers. The system aimed not only to provide an accessible way of expressing programs but also to explicitly teach a model of computation in the process. It used analogies to everyday objects to present a user interface through which the novice programmer expresses what they wish their program to do: the computer's math processor and logic unit are represented by a calculator, a stack of initially blank paper represents memory, a clipboard with worksheets represents the call stack, and so forth. Figure 11.29 shows a screenshot.

The user of Gilligan's system – the programmer – takes on the role of a clerk who intends to accomplish a task using this equipment according to certain rules. In doing so, the user produces a sequence of actions that defines a program. Using the calculator produces an arithmetical or logical expression, for instance, and adding a new worksheet to the clipboard starts a subroutine call (Figure 11.30). The system writes and displays the resulting program as Pascal code that matches the user's interactions with the GUI. By engaging in these clerical activities, the user would learn about their correspondence to the execution model of Pascal programs and would – hypothetically, at least – be better equipped to transition to regular programming later.

The system was never evaluated in practice, as far as I know. Deng (2003) implemented another prototype system which extended Gilligan's ideas to object-oriented Java programming.

ViRPlay3D2

Jiménez-Díaz et al. (2005) presented early work on a 3D environment which was intended to allow students to role-play and learn to understand the (inter)actions of the objects of a given OOP program as it runs. This project has since evolved towards having groups of students role-play object actions in order to *specify* the actions of an OOP program that is to be created. Jiménez-Díaz et al. (2008, 2011) used a system called *ViRPlay3D2* to provide a virtual three-dimensional world in which students control avatars that each represent an object in the execution of an object-oriented program (Figure 11.31). Each student tries to follow the instructions specific to his or her kind of object. One of the students at a time is active and can delegate tasks to other objects by passing messages to them. The goal is to collaboratively produce a working object-oriented software design for a problem and to verify that it works. Students can choose to change the definitions of classes to improve their design. Execution simulations can be saved and reviewed. The system also features a "scripted mode" in which the characters act automatically instead of being controlled by students. In addition to designing programs, ViRPlay3D2 can also be used to examine case studies of program design.

Although ViRPlay3D2's main goals involve high-level software design – in this, it is related to visual programming systems – it also serves as a visualization of a high-level object-oriented notional machine that features classes, objects, and message-passing. Participating within ViRPlay3D2 could help CS1 students learn about the object-oriented execution model.

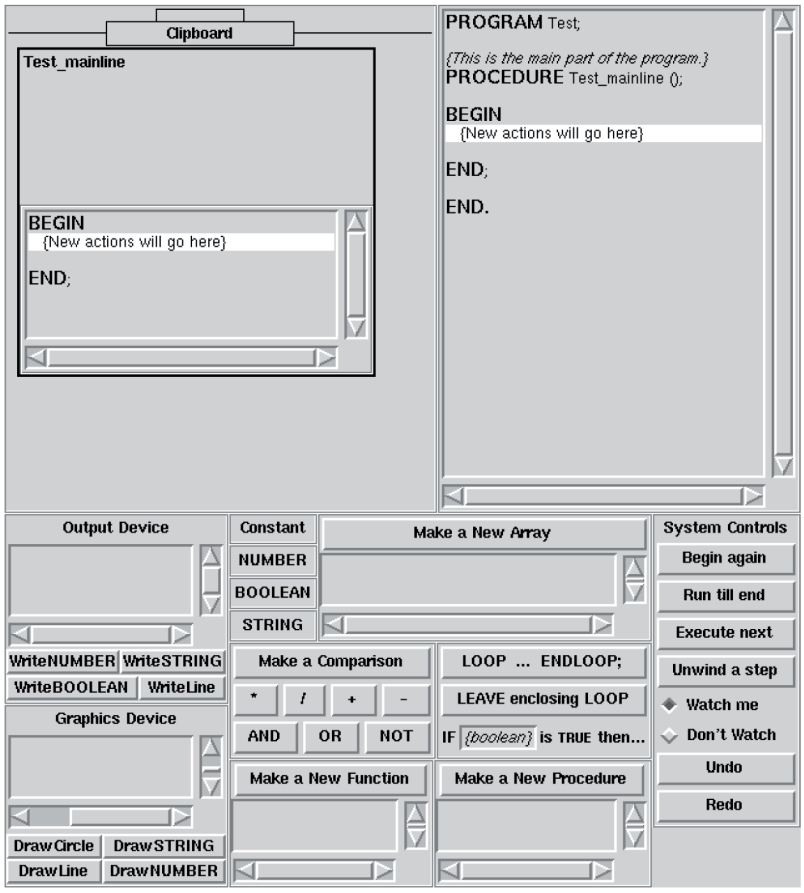


Figure 11.29: Gilligan's (1998) programming-by-demonstration system.

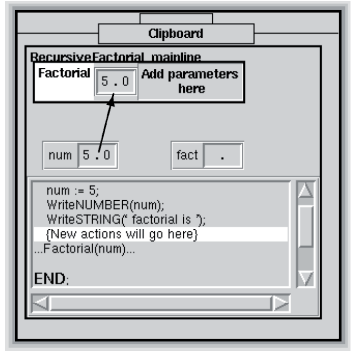


Figure 11.30: Calling a function in Gilligan's system. In this instance, the function is being called for the first time and has not been defined yet. The user carries out the call manually, thereby defining the function by demonstration. He has already created a worksheet (stack frame) named Factorial in the clipboard (stack), thereby adding the line `Factorial()` into the definition of the main program. Just now, he has dragged the value 5.0 from the variable `num` into the frame, adding the parameter expression `num` to the code. The user may now proceed by clicking on the function name to call it, at which point he will start adding code to the function body. For an in-depth explanation of the example, see Gilligan (1998).

Figure 11.31: Students interact within the virtual world presented by ViRPlay3D2 to design object-oriented programs (Jiménez-Díaz et al., 2008). Two separate screenshots are shown here. On the left, a user is viewing message passing in action between two other objects (avatars) from a first-person perspective. Passing a message is graphically represented as throwing a ball to the recipient. On the right, a user is examining another object's Class-Responsibility-Collaboration card (Beck and Cunningham, 1989), which describes how he can interact with the object.

The results of an experimental study (Jiménez-Díaz et al., 2011) did not show a significant difference between ViRPlay3D2 users and a control group that role-played object-oriented scenarios without the help of a software system. Both students and instructors liked the software.

Dönmez and İnceoğlu's tool

Dönmez and İnceoğlu (2008) present a program visualization system that engages students in taking an active role in program execution in order to improve their understanding of the notional machine that underlies C# programming. Using the tool's GUI, students simulate the execution of code they have written in a limited subset of C#. Students use the GUI to evaluate arithmetical and logical expressions, and to create and assign to variables.

Figure 11.32 shows the main display of Dönmez and İnceoğlu's system. The view resembles that of a regular visual debugger, with one major difference: there is no way to make the computer run or step through the program. Instead, GUI controls are present that allow – and require – the user to indicate what happens when the program is run. Here are the steps that the user is expected to follow to deal with the example code in Figure 11.32:

`int sayi;`
Press the New Variable button in the variables area; a dialog pops up (Figure 11.33). Type in the variable name, select its type, and click OK. This makes an icon representing the variable appear in the variables area. Proceed to the next line of code by clicking it.

`Convert.ToInt32(Console.ReadLine());`
The system notices the input function and immediately prompts for input. Enter a string; it appears in the literals area. Drag the input into the active area to be processed and select `Convert.ToInt32` from the Functions menu to produce the return value within the active area. Then drag the return value into the variable `sayi` before moving on to the next line.

`sayi = sayi + 20;`
Calculate the sum by dragging the variable value to the active area, selecting the sum radio button, and dragging the literal value 20 from the literals area (where it has appeared

automatically) to the active area. The result now appears automatically in the active area. Drag it into the variable and click the last line.

`Console.WriteLine(sayi)`
First drag the parameter value into the active area, then select the function from the Functions menu as above.

Dönmez and İnceoğlu's system can only handle certain simple kinds of programs. Many fundamental topics with complex runtime dynamics – such as functions (except for a handful of built-in single-parameter functions, as above), references and objects – are not supported.

To my knowledge, no evaluation of the system has been reported.

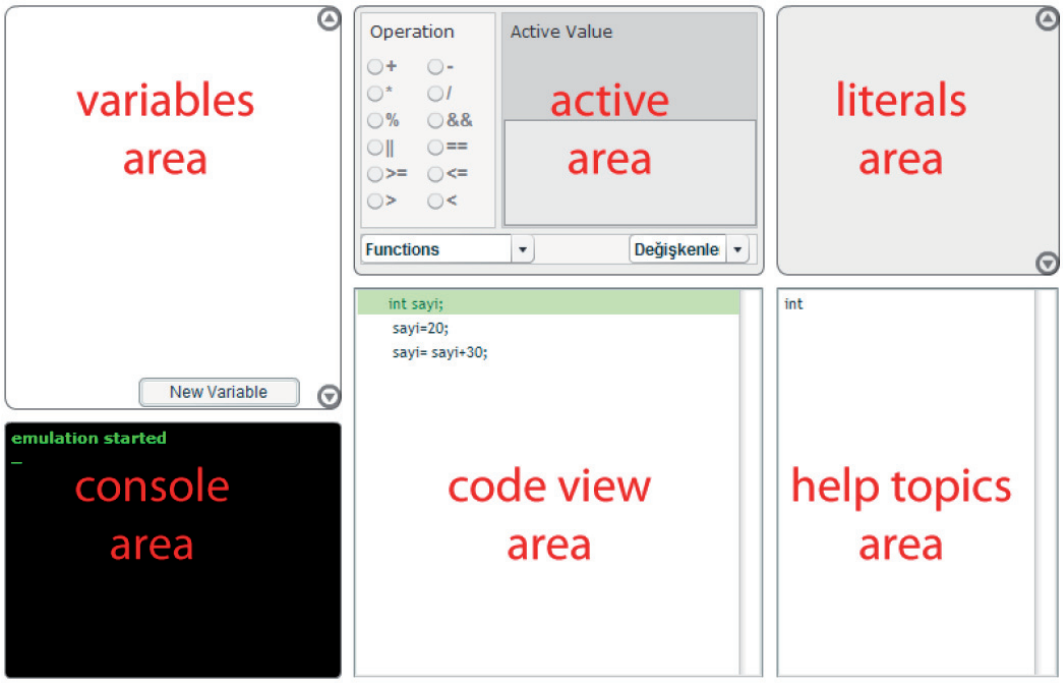


Figure 11.32: Dönmez and İnceoğlu's (2008) program simulation system.

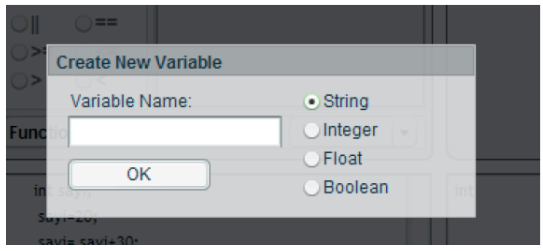


Figure 11.33: Creating a variable in Dönmez and İnceoğlu's (2008) system.

Online Tutoring System

Kollmansberger (2010) developed and used another highly interactive system. His *Online Tutoring System* (Figure 11.34) presented students with short ready-made programs written in Visual Basic for Applications. Similarly to Dönmez and İnceoğlu's tool, the Online Tutoring System required the student to predict which statement the computer will execute next (by clicking on the correct line), and to indicate the order in which the parts of the statement are dealt with (by clicking on the appropriate part of the code). Furthermore, the student must type in the resulting values at every stage of expression evaluation and assignment (examples in Figures 11.34 and 11.35). The system provided textual prompts at each step of the way to indicate what type the next step should be (e.g., subexpression selection or accessing a variable).

When the student got an answer wrong, the system told them the correct answer, which the student nevertheless had to reproduce within the system in order to proceed. Incorrect answers were reflected in the student's grade for the assignment, but the same assignment could be tried repeatedly.

The Online Tutoring System supported a subset of Visual Basic that included function calls, but excluded recursion and object-oriented concepts, for instance.

Kollmansberger (2010) reported that two sections of a CS1 class that used the Online Tutoring System for twelve additional exercises at various points of the course had an average course completion rate of 92%. For the four sections that did not do these additional exercises the corresponding figure was only 66%. According to opinion surveys, the students liked it for the most part, although some found the user interface unwieldy and the exercises too repetitive.

Clouds & Boxes in ViLLE

Recent versions of the ViLLE system (see above) feature a new assignment type – "Clouds & Boxes" – which is inspired by the present work on visual program simulation. In a Clouds & Boxes assignment, ViLLE does not run the program automatically; instead, the student has to manually carry out certain aspects of program execution themselves in order to make execution proceed. The focus is on variables, assignment, and the use of stack frames.

For instance, the program in Figure 11.36 would be correctly simulated as follows (Rajala et al., n.d.):

> `int a = 3;`
> Click *New variable* to make a dialog appear. Click on a *Basic variable* button in the dialog, choose `int` from a pulldown menu, type in the name of the variable and its initial value in text fields. Press the *Step* button to move to the next line.
>
> `System.out.println(myFunction(a));`
> Nothing to do here at this point, simply press *Step* to move to line 5 inside the method.
>
> `public static int myFunction(int parameter) {`
> Click *Subprogram call* to create a new frame on the stack. Type in the name of the function being called and the number of parameters. Type in the name of the parameter and its value. Click *Step* again.
>
> `return parameter * 2;`
> No action needed, returning is automatic. Click *Step*.
>
> `System.out.println(myFunction(a));`
> Back on line 3. Finish the call by clicking *Remove from stack*. You are done.

ViLLE's Clouds & Boxes assignments are at a prototype stage; at the time of writing, the results of any empirical evaluations have not, to my knowledge, been published.

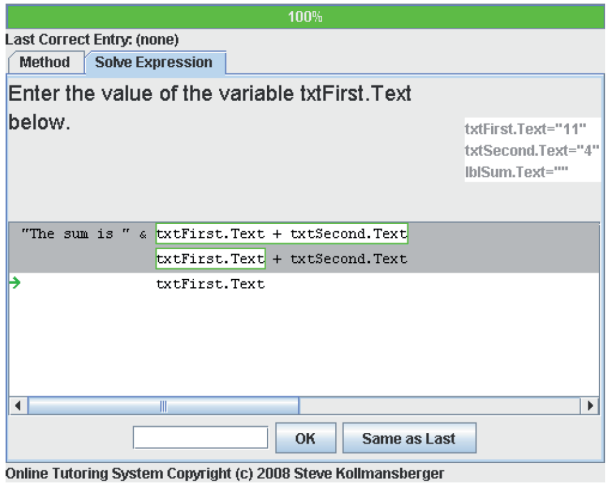


Figure 11.34: An assignment on string catenation within the Online Tutoring System (Kollmansberger, 2010). The student has first chosen the expression `txtFirst.Text + txtSecond.Text` for evaluation, then the subexpression `txtFirst.Text`. Now the system informs them that the next step is to access the variable: consequently the user has to look at the values in memory (on the right) and type in `"11"` (in quotes, to highlight that it is a string). Subsequent steps include typing in `"4"` and `"114"` at the appropriate moments.

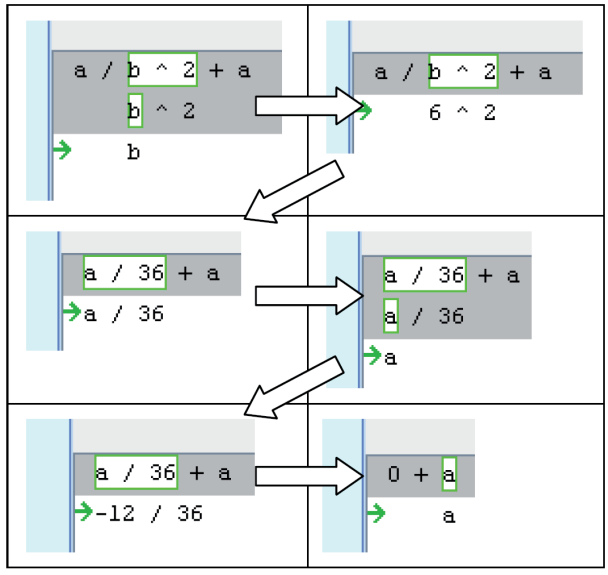


Figure 11.35: Stages of expression evaluation within the Online Tutoring System (Kollmansberger, 2010). Some steps have been omitted.

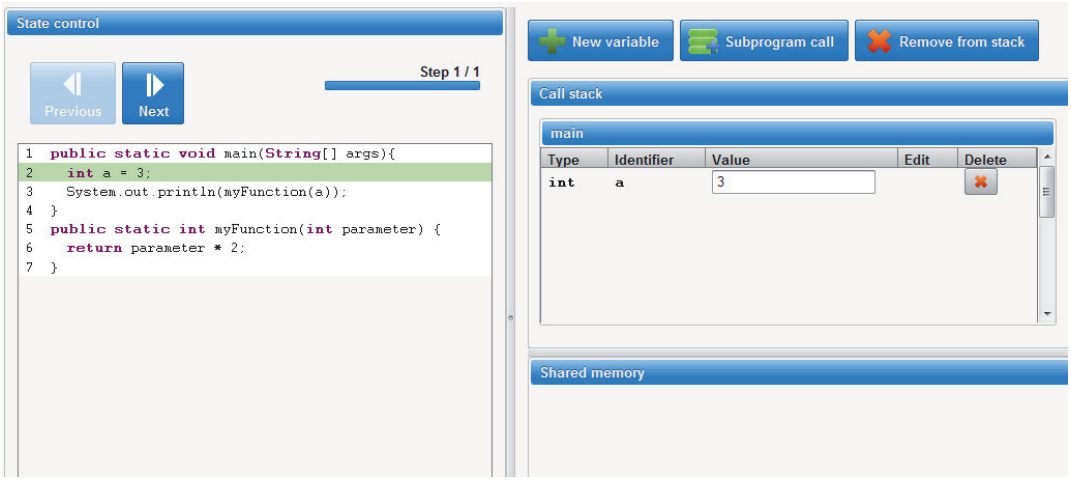


Figure 11.36: A "Clouds & Boxes" assignment in ViLLE has the user take charge of running a program. The buttons at the top allow the user to manipulate variables and the call stack.

11.3.4 Various systems take a low-level approach

The systems reviewed in this chapter so far have represented notional machines that operate on high-level languages at various levels of abstraction that are not very close to the underlying hardware. One way to learn about program dynamics – albeit rare in CS1 – is to first build low-level foundations by visualizing an actual or simplified system that explains programming on the assembly language or bytecode level. Many existing program visualization tools might serve a purpose in such an endeavor.

Biermann et al. (1994) created a system – *This is How A Computer Works* – for visualizing Pascal programs to freshmen at different levels of abstraction below the code: a compiler level, a machine architecture level, and a circuit level. The simulator *MPC1* presented a simplified computer in terms of processor operation codes, RAM cells, and registers (Sherry, 1995). The *SIMPLESEM Development Environment* (Hauswirth et al., 1998) and *MieruCompiler* (Gondow et al., 2010) also map high-level language semantics to assembly code. *EasyCPU* (Yehezkel et al., 2007, and references therein) visualizes a model of computer components – registers, I/O ports, etc. – and highlights how each component is active in the execution of an assembly-language program. A different sort of low-level approach was implemented in *ITEM/IP*, which visualized the execution of a Pascal-like language in terms of a Turing machine (Brusilovsky, 1992).

Some low-level systems have featured modes of interaction beyond controlling the view. In the *CpuCITY* virtual 3D world the student used an avatar to perform hardware-level operations such as "take this packet to RAM" (Bares et al., 1998). *JV²M* applies a similar immersive approach to Java programming by having a student-controlled avatar perform in-game tasks that match the bytecode instructions corresponding to a given Java program (e.g., Gómez-Martín et al., 2005; Gómez-Martín et al., 2006). Figure 11.37 shows a screenshot of JV²M. The way in which CpuCITY and JV²M engage the student makes them similar in a sense to our concept of visual program simulation (see Part IV), although the level of abstraction and the virtual world approach are quite different.

Various other low-abstraction PV systems exist; the above is a fairly arbitrary selection. Many of the systems may be useful for the goal of teaching about low-level phenomena (which is indeed the stated goal of most of the systems). I will not review low-level systems in more detail, as my primary interest is in CS1 and systems that help students learn the runtime semantics of a high-level language. I consider systems that deal with assembly language too detailed to be practical for this purpose – the 'black boxes' inside

Figure 11.37: The user is executing a compiled Java program in JV^{2}M (image from Gómez-Martín et al., 2006). The bytecode instructions correspond to actions in the virtual world. The user's avatar is the big feline in the middle. The dog on the right is Javy, a helpful intelligent agent that provides advice. I do not know what the mouse is.

the 'glass box' of the notional machine shown are too small and too numerous to achieve the necessary simplicity (cf. du Boulay et al., 1981).[8]

11.4 In program visualization systems for CS1, engagement has been little studied

Many of the visualization systems for CS1 have been evaluated informally, with positive teacher experiences and encouraging course feedback often being reported. Some of the systems have also been evaluated in more rigorous ways, either qualitatively or through controlled experiments. Usually, the evaluations have been carried out by the system authors themselves at their own institutions, often in their own teaching. Many of the findings from these studies have been positive, suggesting that the program visualizations have served a purpose, at least in the context in which they were crafted.

What about the role of engagement, identified as potentially so important by the software visualization community (Section 11.2 above)? As Table 11.6 (above on p. 154) shows, most of the systems within the scope of my review are meant for controlled viewing of given or student-created content, but a few recent systems especially feature other ways of engaging students. To what extent do the hypothesized engagement effects apply to notional-machine visualization?

Barely any of the empirical studies of the program visualization tools that I have reviewed have been phrased in terms of the engagement taxonomies from Section 11.2.2. However, we may consider whether the existing research answers any of the questions regarding modes of engagement. In Section 11.3, I referred to a number of studies that have evaluated the various program visualization systems. Table 11.7 uses the 2DET to summarize those of the evaluations that sought to determine the relative effectiveness of different engagement levels. To produce Table 11.7, I made the following delimitations.

[8] I do think that showing compiled code to students in CS1 can be instructive, but as an additional measure, not as the main way to learn about a notional machine for a high-level paradigm.

Table 11.7: Experimental evaluations of program visualization tools comparing levels of engagement.

System	Source	Scope of experiment	Treatment	Control	Measure	Statistical significance?
two visual debuggers	Bennedsen and Schulte (2010)	lab	controlled viewing / given content	no viewing / given content	reading tasks	no
Bradman	Smith and Webb (2000)	lab	controlled viewing / given content	no viewing / given content	reading tasks	yes (in one task only)
VIP	Ahoniemi and Lahtinen (2007)	lab	controlled viewing / given and modified content	no viewing / given and modified content	writing tasks	yes (among novices and strugglers)
ViLLE	Rajala et al. (2008)	lab	controlled viewing (and responding?) / given content	no viewing / given content	reading and writing tasks	no
ViLLE	Kaila et al. (2009a)	lab	responding / given content	no viewing and controlled viewing / given content	reading and writing tasks	yes
ViLLE	Kaila et al. (2010)	course	responding / given content	no viewing / given content	reading and writing tasks	yes

- The table lists only quantitative between-subject experiments that checked for the statistical significance of results.
- Only studies that used different modes of engagement as conditions are listed. (Not using a visualization at all does count as no viewing.) This excludes studies comparing two different tools (e.g., a regular visual debugger vs. an educationally oriented visualization) used in the same way.
- Only studies in which the participants learned about typical CS1 content are included.
- Studies that gave the experimental group additional assignments to do on top of those of the control group are excluded.
- Only studies in which researchers or teachers assessed learning outcomes are included. This rules out opinion polls, for instance.
- Only studies in which an intervention was followed by a distinct assessment phase that is the same for each group (e.g., an exam) are included.

Table 11.7 shows that most of the experimental studies to date involve comparisons between low levels of engagement, and usually pit controlled viewing or responding against not using a visualization at all. The studies have revolved largely around example-based learning and given content. Most of the systems built have not been experimentally evaluated at all.

The results of these evaluations largely support the use of visualization, but it is difficult to draw further conclusions. All in all, the CER literature does not yet tell us very much about the relative effectiveness of levels of engagement beyond controlled viewing in the context of program visualization for CS1.

Part IV

Introducing Visual Program Simulation

Introduction to Part IV

We have now seen that there are serious problems in introductory programming education (Part I), that there is significant concern about novice programmers' difficulties with understanding program dynamics and the notional machines that run programs (Part II), and that program visualization is one way to teach about this challenging topic (Part III).

In Part IV, I formulate the idea of visual program simulation (VPS), a pedagogical technique for learning about program execution.

Chapter 12 is a brief introduction to visual program simulation. Chapter 13 presents a new tool, UUhistle, that lends software support to VPS. In Chapter 14, I draw on the literature presented in Parts I to III to explain why VPS is a sensible approach in the light of what is known about introductory programming education. Finally, Chapter 15 explains some of the specific design decisions taken during UUhistle's development.

All this lays the foundations for the empirical evaluation of VPS in Part V.

Chapter 12

In Visual Program Simulation, the Student Executes Programs

This short chapter describes the concept of *visual program simulation*, the interactive visualization-based approach to teaching about program dynamics that I will investigate in the rest of this thesis.

In this chapter, I merely outline what visual program simulation is. Later chapters will provide more detail and justify the approach in relation to learning theory and other prior work.

12.1 Wanted: a better way to learn about program dynamics

Here is my agenda in brief:

- to help novice programmers understand the notional machine that they are learning to control through programming;
- to help novice programmers reason about programs' runtime behavior and make successful predictions about what programs do, and
- to help novice programmers learn about the semantics of programming constructs.

And here is how our approach addresses the goals:

- by visualizing a notional machine;
- by having novice programmers explicitly and directly control the notional machine through the visualization, and
- by encouraging students to be cognitively active as they trace a program's execution.

12.2 Visual program simulation makes learners define execution

Visual program simulation (VPS) is an activity that immerses students in the dynamics of program execution.

In visual program simulation, a learner takes on the role of the computer as executor of a program. The learner has to 'do the computer's job': read code, follow instructions in the appropriate order, make control flow decisions, and keep track of program state. To keep things explicit and manageable, the learner uses a visualization of some kind as an external aid. The visualization – a drawing on paper or a software visualization tool, for instance – helps the learner to track state changes and reason about program behavior.

Visual program simulation can serve as an exercise in which learners practice tracing programs and learn about a notional machine. In a VPS exercise, the learner not only has to think about how a program works, but also to demonstrate their understanding in a concrete way – at least to themselves, and perhaps to a teacher as well. Ultimately, albeit indirectly, visual program simulation is intended to contribute to the skill of writing programs.

An example

Here is a short Python code fragment that defines a function and calls it:

```
def square(x):
    return x * x

number = 5
result = square(1 + number)
```

Let us consider how a programmer might visually simulate the execution of the last two lines. The simulation could involve the following steps, performed by the learner while reading the code.

1. The learner makes a visualization of the variable `number` and marks it as having the value 5.
2. The learner determines the value of the parameter expression by adding 1 to the value of `number`, and makes a note of this result (6).
3. The learner makes a visualization of a call stack frame.
4. The learner adds to the frame a visualization of a variable called `x`, taking a value (6) for it from the evaluation in Step 2.
5. The learner determines the return value (36) and makes a note of it.
6. The learner makes a visualization of a new variable `result` alongside `number` and marks it as having the returned value. (The frame created in Step 3 becomes redundant.)

In practice, 'making' visualizations could mean either drawing them from scratch or using ready-made components in a suitable software tool such as the one I describe in the next chapter. In Section 14.4, I will argue that tool support increases the potential of visual program simulation.

What I just presented is one way of simulating the execution of the example code. Other ways can include more steps (e.g., the learner also explicitly keeps track of the current line), have a different overall level of abstraction (e.g., a bytecode level), or present the execution from a different perspective (e.g., a functional one based on term rewriting). Different kinds of notional machines and visualizations can be used as the basis for visual program simulation. The defining features of VPS are the driving role of the learner and the use of an external visual representation as an aid.

12.3 Visual program simulation is not program animation or visual programming

Let us consider the relationship VPS has to a few related concepts and terms.

Visual program simulation is a type of *program tracing*, a term that I use in a generic sense to mean examination of the execution sequence of a program. Tracing can be done by people, computers, or both in collaboration. For instance, a visual debugger traces the execution sequence of a program and helps the programmer to mentally trace the program.

I briefly defined software visualization and its main subfields at the beginning of Chapter 11. In this scheme, visual program simulation falls under the domain of software visualization, more specifically program visualization, as it is concerned with concrete programs rather than generic algorithms (see Figure 11.1 on p. 142).

Again in terms of the classification from Chapter 11, a visual debugger is a *program animation* system. In program animation, the computer determines what happens during a program run, and visualizes this information to the user. The key difference between program animation and visual program simulation is just this: in the former it is the computer that automatically defines what happens and when, while

VPS leaves this task to the user. Most of the educational systems reviewed in Chapter 11 are program animation systems.[1]

Visual programming is a form of program visualization that is related to, but distinct from, visual program simulation. Visual programming shares with VPS a high level of user interaction with a program visualization. However, the purpose of visual programming – to use a visualization as a means for program creation – is quite different from that of VPS, which is a vehicle for exploring the behavior of existing programs.

[1]My use of the terms 'simulation' and 'animation' follows that of Korhonen (2003), who similarly distinguished between *algorithm animation* and *visual algorithm simulation* in his work on interactive algorithm visualization; see Section 13.8.

Chapter 13

The UUhistle System Facilitates Visual Program Simulation

UUhistle (pronounced "whistle") is a program visualization system for introductory programming education. It is meant for visualizing the execution of small, single-threaded programs in CS1 courses and similar settings.

UUhistle visualizes a notional machine for the Python programming language. This notional machine executes programs on a level of abstraction right below program code, in terms of expression evaluation, the call stack, variables, objects, classes, references, and other concepts.

UUhistle can be used for a number of activities, including the following.

- *Debugging with animations*. Learners can view animations of the execution of programs that they wrote themselves. In this mode of use, UUhistle serves as a graphical, very detailed sort of debugger.
- *Exploring examples*. Learners can view animations of teacher-given programs. Teachers may configure UUhistle to adjust how particular example programs are shown.
- *Visual program simulation*. Learners can engage in visual program simulation exercises in which they manually carry out the execution of an existing program (see previous chapter). UUhistle facilitates VPS by providing visual elements for the learner to manipulate. UUhistle gives automatic feedback on VPS exercises and can automatically grade students' solutions.
- *Interactive program animation*. UUhistle can animate the execution of programs created on the fly so that the execution of each instruction is shown as soon as the instruction is typed in. This allows for another interactive, graphical way of exploring program behavior.
- *Presentations*. Teachers and learners can use UUhistle's animations as an aid when explaining the execution-time behavior of programs.
- *Quizzes*. Stop-and-think questions can be embedded into teacher-defined example programs for learners to answer.

The primary feature that sets UUhistle apart from the mainstream of educational program visualization is the support the system provides for visual program simulation. This chapter is a tour of UUhistle's main functionality, including but not limited to VPS. The later chapters in this thesis revolve around VPS.

Section 13.1 below introduces UUhistle's GUI and its animations of program execution. Section 13.2 shows how the system can interactively animate programs created on the fly. Section 13.3 is an overview of the options available for the teacher to configure program examples for students to view later. Section 13.4 describes how visual program simulation works in UUhistle. Section 13.5 briefly presents how the system supports automatic grading. Section 13.6 reviews the different ways in which UUhistle allows learners to engage with program visualizations. Section 13.7 is an advertisement for the UUhistle web site, and Section 13.8 concludes the chapter by comparing UUhistle to its closest relatives.

The UUhistle system has been developed as a part of the research project reported in this thesis. It is joint work by Teemu Sirkiä and myself. I have been responsible for the conceptual design of the system,

specific requirements, and the embedded textual materials. Teemu has done the software design and the coding. We have both contributed to the user interface.[1]

UUhistle supports a sizable, and growing, subset of the Python language. This thesis does not document all the capabilities of the system.

13.1 UUhistle visualizes a notional machine running a program

Figure 13.1 shows a snapshot of the most recent UUhistle release, paused during the execution of a Python program. The right-hand side of UUhistle's display is dominated by a visualization of the state of a notional machine. Contents of memory appear as abstract graphics.

What follows is an overview of the main features of UUhistle's visualization and user interface.

13.1.1 State is shown as abstract graphics

UUhistle displays the program code of a running Python program on the left (marked 1 in Figure 13.1). The current line is highlighted by a blue background and a blue arrow pointing at it.

Both the trivial and composite objects stored within the heap appear in the top right-hand corner. Objects of user-defined types such as `Car` in Figure 13.1 have explicit object identifiers and instance variables. Objects, like many things in UUhistle, are visualized as rounded, colored rectangles – objects are blue (with a few exceptions, see below). Objects of certain common immutable Python data types – `int`s, `float`s, `boolean`s, and `string`s – as well as certain exception types have simplified default representations, as also shown in Figure 13.1. This reduces visual clutter and allows simple Python programs to be visualized in a way that does not rely on object-oriented concepts. Below, I will occasionally refer to objects of these types as "simple values".

Classes and functions appear in their own panels (3 and 4 in Figure 13.1). Although Python classes and functions are also objects in the heap, this division emphasizes the roles that class and function objects play in most Python programs. In this example, we have a user-defined class `Car` and two built-in functions. To save space and simplify the visualization, UUhistle seeks to display only those classes and functions that the running program directly uses; the same goes for operators (5). As it is not possible to fully know in advance which classes and functions will be needed, UUhistle uses the program code and some simple heuristics to make a 'best guess'. If it turns out during the program run that more classes or functions are needed, those are added on the fly. (This does not commonly occur in CS1 example programs.)

The call stack and its frames (6) are where most of the action is. In Figure 13.1, the method `add_fuel` has been called from the module level and the body of `add_fuel` is being executed. Consequently, there are two frames: one for the module level and another for the call. Each frame has its own expression evaluation area. The ongoing evaluation of `work_car.add_fuel(10)` is shown within the bottom frame, and another expression is being processed in the upper frame. Each stack frame may also contain variables; UUhistle visualizes all variables as rounded green rectangles, irrespective of their contents and location in memory. Built-in functions and methods are treated as 'black boxes' – their internal behavior is not visualized.

The representation of a references features the text "ref" and an object identifier. Whenever the mouse cursor is on top of a reference, the reference is additionally visualized by an arrow pointing to the target object. To underline to the learner that multiple references can point to the same object, UUhistle also shows other references with the same target as dimmer arrows. An example is shown in Figure 13.1.

You can change a few aspects of the visualization through the Settings menu or with hotkeys. For instance, you can toggle whether simple values are shown as in Figure 13.1 or as full-fledged objects. If you find the dimmed 'sibling arrows' of references (see above) distracting, you can disable them.

[1]Parts of this chapter are taken from two earlier publications. 1) Juha Sorva and Teemu Sirkiä: "UUhistle – A Software Tool for Visual Program Simulation", In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 49–54, ©2010 Association for Computing Machinery, Inc. `http://doi.acm.org/10.1145/1930464.1930471` Reprinted by permission. 2) Juha Sorva and Teemu Sirkiä: "Context-Sensitive Guidance in the UUhistle Program Visualization System", In: *Proceedings of PVW 2011, Sixth Program Visualization Workshop*, pages 77–85, Darmstadt, Germany, Technische Universität Darmstadt, 2011. Reprinted by permission.

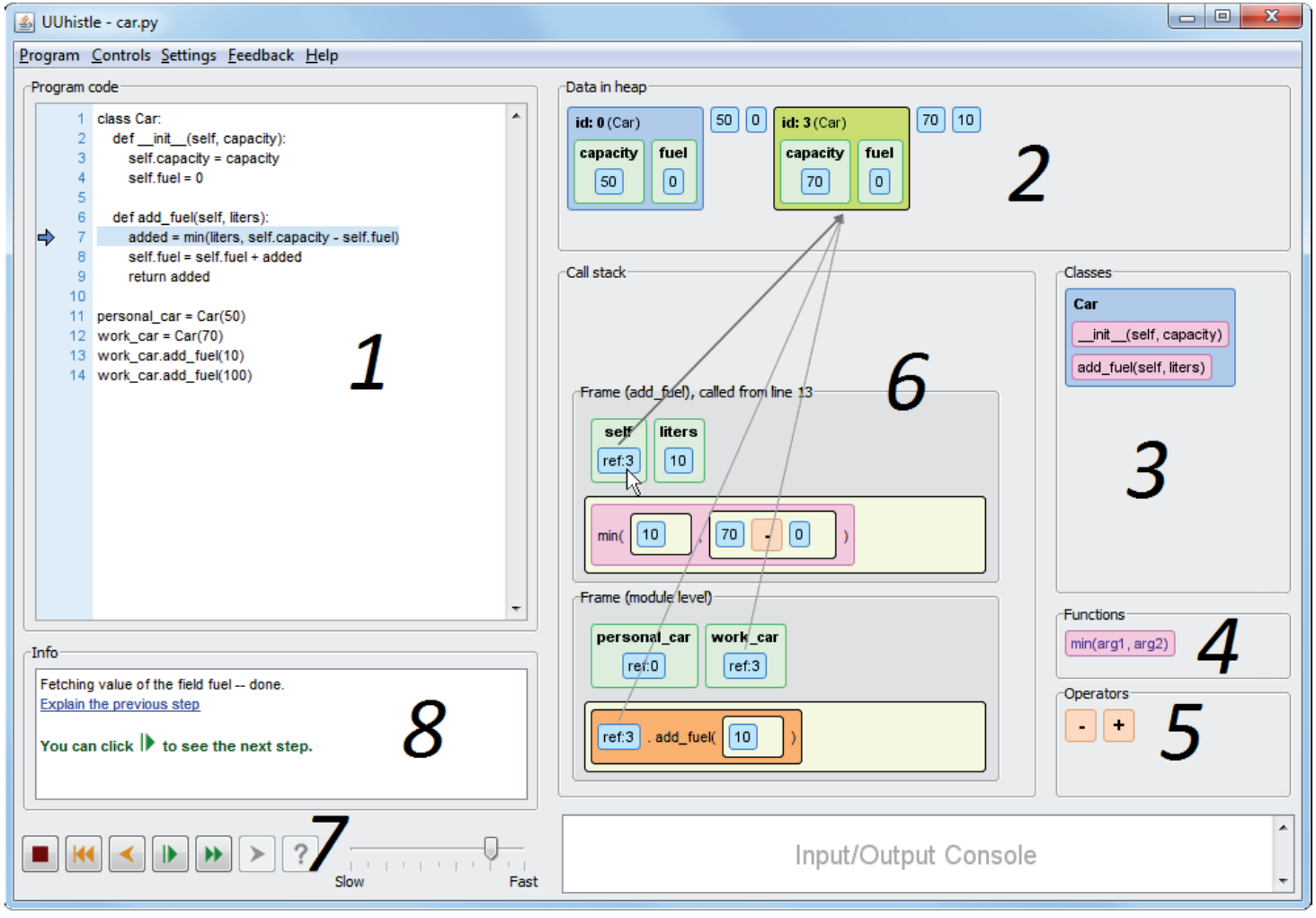


Figure 13.1: An animation of a Python program in UUhistle v0.6. The program is paused on line 7, just before the subtraction is carried out.

You can adjust the sizes of various panels and the overall size of the GUI window with the mouse. If display space runs out, scrollbars appear automatically.

13.1.2 It is simple to watch an animation

In the default mode, UUhistle serves as a program animation system that takes a Python program and shows how it executes. How do you get UUhistle to animate a program?

First, you need a program

The Program menu provides a selection of options for creating Python programs and accessing existing ones. You can start editing a blank file. You can load an example program, previously created and configured by a teacher. Or you can open an existing Python source code file from the local file system.

UUhistle features a basic text editor for creating and modifying single-file programs. The editor is available on the left of the UUhistle window whenever no program run is being visualized. The editor supports standard GUI operations; you can, for instance, paste in Python code from another source.

Teachers can define ready-made example programs, which you can then load into UUhistle either online or locally (see Section 13.3 below).

Much as in any modern visual debugger, you start running the program using either a control button, a menu item, or a hotkey.

Then, you control its viewing

Once the program run has been started, you choose the pace of execution and may step back and forth within the execution sequence as you please.

To control the viewing of the program's execution, you press the buttons in the lower left-hand corner (near number 7 in Figure 13.1). From left to right, the buttons are: Stop, Rewind, Undo, Next Step, Next Line, Redo, and Hint. Of these, the Next Step button is the most central to program animation in UUhistle: pressing the button tells UUhistle to show the next execution step and pause again afterwards. Choosing Next Step repeatedly lets you see the entire program executed in small steps.

UUhistle uses smooth animations to draw attention to what happens. To the right of the control buttons is a slider that adjusts the speed of these animations.

Execution steps: an example

The execution of a single line of code usually consists of several steps. As an example, the execution steps of the Python statement `b = a + 1` are listed below, and illustrated in Figure 13.2.

1. The value of `a` is fetched from memory, shown as a copy of the value 'gliding' from the variable to the expression evaluation area in the topmost frame.
2. The operator + is fetched from memory, shown as a copy of the operator gliding from the Operators area to the expression evaluation area.
3. The simple value `1` is fetched from memory, shown as a copy of the value gliding from the "Data in heap" panel to the expression evaluation area. (If the value did not already exist in the heap, it instantly appears there first.)
4. The sum is calculated, indicated by the operator being briefly highlighted, after which the resulting value replaces the arithmetical expression in the expression evaluation area.
5. The sum glides from the expression evaluation area into the variable `b`, which is created in the process if it did not exist already. A new variable that is being created is first shown in outline while the value glides towards it, then in full once the value reaches its destination.[2]

After these steps are performed, control moves from the current line to the next, as a separate execution step.

[2]N.B. UUhistle's notional machine reflects the Python language, in which variables are bindings of names to objects. In this abstraction, a variable does not exist before it is first bound to a value.

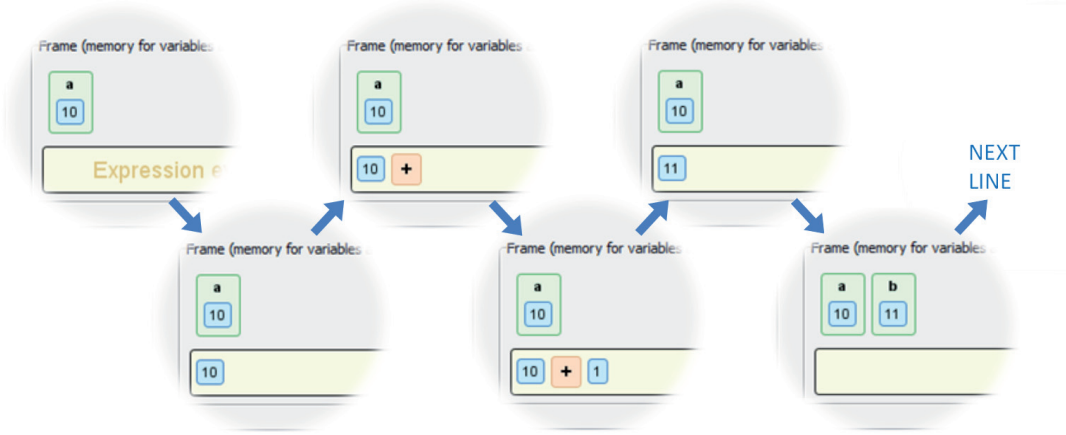


Figure 13.2: The between-step stages of the execution of `b = a + 1`, as animated by UUhistle. Pressing Next Step causes UUhistle to transition to the next stage. Most transitions are accompanied by a smooth animation.

More GUI controls

The Next Line shows all the (remaining) steps of the current line as one continuous animation at a high speed, then pauses execution. You may keep the Next Line button pressed to fast-forward an arbitrary number of steps.

The Stop button terminates execution. Rewind restarts it. Undo reverts to the state at the previous execution step or line, without reverse animation, canceling an earlier Next Step or Next Line command. Redo cancels one such backstep, also without animation. The Hint button is only relevant to certain kinds of visual program simulation exercises (Section 13.4 below).

The primary way to view short program animations in UUhistle is one step or line at a time. However, sometimes you want to get past a section and view a particular stage of execution. Clicking on a line of code (or its line number) brings up a context menu in which you can choose to continuously execute the program until that line is reached or there is another reason to pause execution, such as a breakpoint. Breakpoints can be set from the same context menu.

Active user control of execution is the norm, and is recommended in order to avoid mere passive viewing. However, you can completely automate execution through the Settings menu. If you do, the Next Step button is replaced by a Play button, which makes the animation run continuously, starting each step after the previous one finishes until you click on the button again to pause.

13.1.3 Textual materials complement the visualization

There is a fairly large amount of textual materials embedded in UUhistle that explain programming concepts and UUhistle's visualization of them, and provide various kinds of hints to the learner. These materials can be accessed through the Info box and through context menus that pop up when you click on visual elements.

The Info box

During and after each execution step, the Info box (number 8 in Figure 13.1) gives a brief status report. While execution is paused, the Info box suggests possible user actions and provides context-specific links to learning materials that UUhistle displays in popup dialogs.

A staple among Info box links is "Explain the previous step" (see, e.g., Figure 13.1). Clicking on this link brings up a more detailed hypertext explanation of the previous execution step (or steps, if multiple

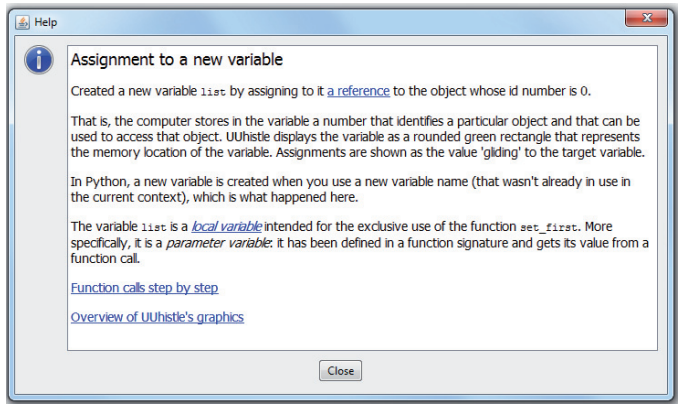


Figure 13.3: An example of a dialog that pops up when the user clicks an "Explain the previous step" link in the Info box. The user can follow links within the dialog to further materials.

execution steps have just been shown in sequence). Many of the explanations are context-sensitive and incorporate knowledge of the specific program run. An example is in Figure 13.3.

Various other links appear in the Info box at certain times during a program animation, allowing learners to read about interesting aspects of program execution. For instance, when multiple references point to an object whose state has just changed, UUhistle provides a link to an explanatory text, as shown in Figure 13.4. When two variables with the same name appear in different stack frames, a link gives access to an explanation of how the local variables of each frame are separate from those in other frames, and how this allows for 'namesakes'. When a function is called for the second time, creating a new frame, a link leads to a text that underlines the difference between a function definition and a function call, and explains why it is not practical to keep in memory a single reusable frame corresponding to each function definition (as students sometimes suggest).

If you click Next Step many times in quick succession – perhaps in an attempt to reach a section of code as fast as possible – a link appears in the Info box to hint about some potentially useful GUI features (Figure 13.5).

"What is this?"

Many elements of UUhistle's visualization can be clicked to bring up a context menu that features a "What is this?" option. Choosing this menu item brings up an explanation of that visual element and corresponding programming concepts. An example is shown in Figure 13.6.

13.1.4 Graphical elements are added as needed

Figure 13.1 shows UUhistle running a program that uses functions, classes, and objects so that they are all treated as 'glass boxes'. However, UUhistle reveals these parts of the notional machine only as they are needed. Specifically:

- the Functions panel is only shown if the program to be visualized contains function definitions, or if any functions are called from the visible code;
- the Classes panel is only shown if the program to be visualized contains class definitions, or if any methods are called, or if classes are used as class objects (e.g., values are assigned to their instance variables);
- the Call Stack is only shown if the Functions or Classes panel is. An additional requirement is that at least one non-built-in function or method is called. That is, there must be a need to visualize the internal details of function or method calls. If not, only a single frame is shown.

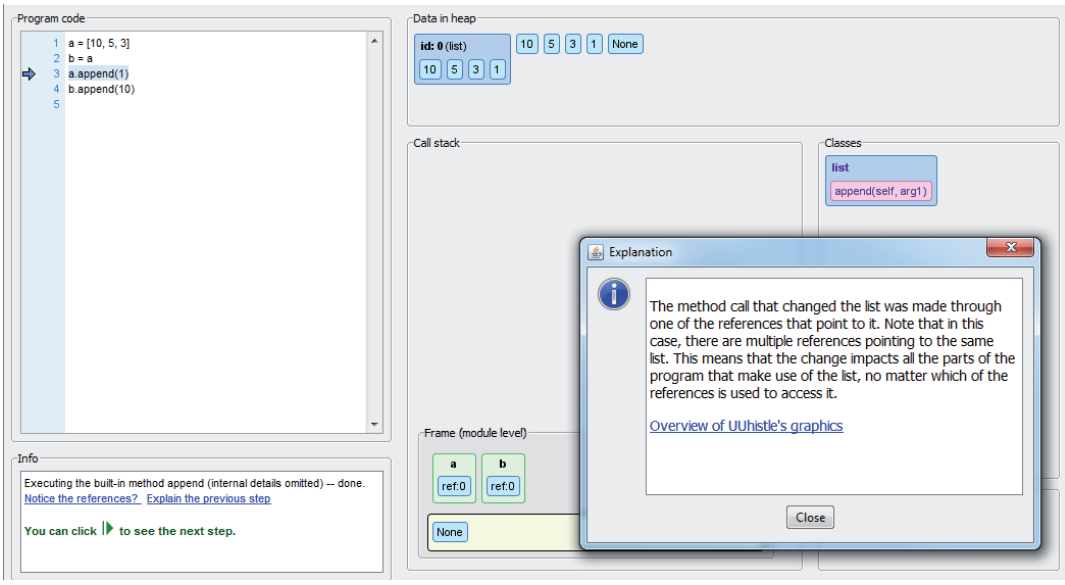


Figure 13.4: UUhistle highlights a point of interest. The "Notice the references?" link gives access to the explanatory dialog.

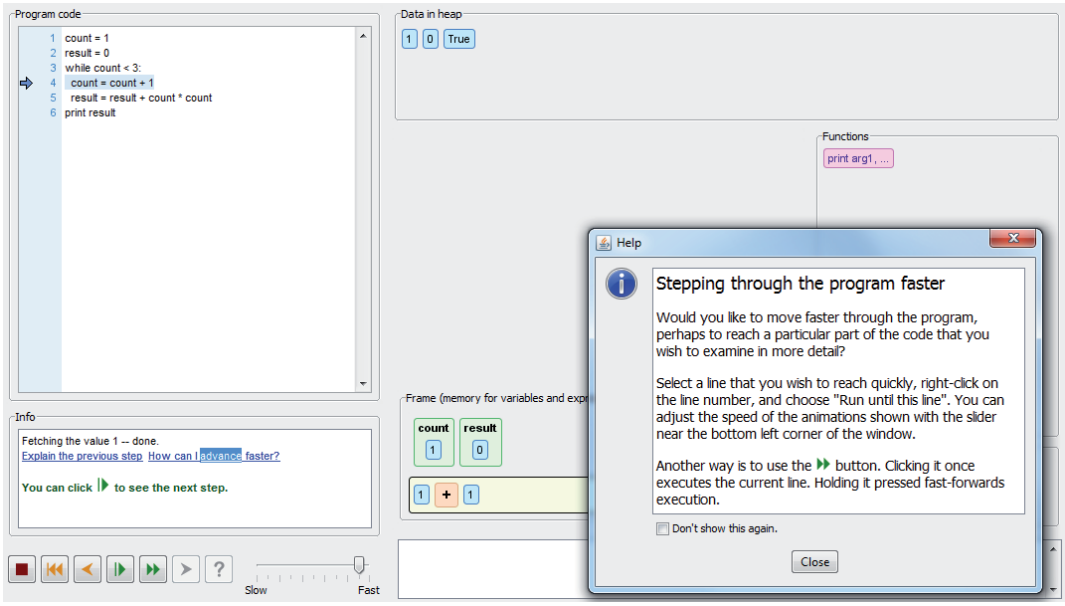


Figure 13.5: If the user appears to be trying to get past a section of code very quickly, UUhistle attempts to help. The dialog appears when the user clicks on the link in the Info box.

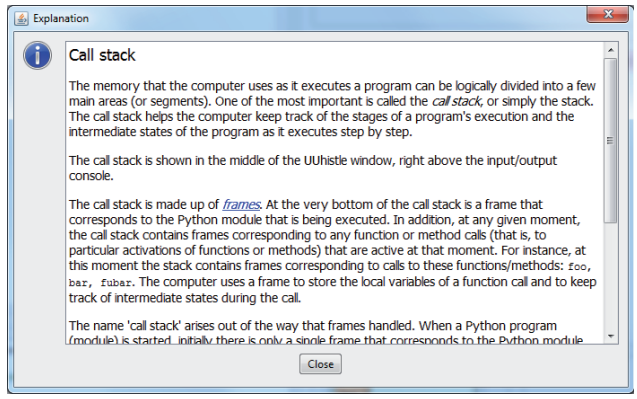


Figure 13.6: An example of a UUhistle dialog that explains a programming concept and UUhistle's visualization of it.

Various examples of simplified views, in which some of these parts have been omitted, appear in the screenshots within this chapter.

There are three interrelated reasons for these simplifications:

- avoiding unnecessary visual clutter and distractions;
- enabling learners and teachers who are so minded to cover certain topics while steering clear of others (e.g., imperative programming without explicit objects), and
- facilitating instructional design that reveals complexity only gradually, starting out with fewer concepts and ending up with more.

This behavior conforms to the consume-before-produce principle (Caspersen, 2007, and references therein), which encourages a form of black-box scaffolding (Hmelo and Guzdial, 1996). Learners can learn how function calls (opaque 'black boxes') are set up before they have to worry about how function implementations work. The same goes for objects. Moreover, UUhistle allows teachers to include 'hidden' functions and methods in predefined program examples (see Section 13.3 below); those are then treated as black boxes, just like functions and methods built into Python. This further strengthens UUhistle's support for the consume-before-produce teaching approach.

The omission of parts of the visualization is reflected in the textual learning materials as well. For instance, when only a single frame is shown rather than a call stack, UUhistle's dialogs explain what a frame is in a simpler way.

13.2 In interactive coding mode, the user can view as they type

The previous section described how UUhistle can animate existing programs that are first written in the UUhistle editor or loaded from a file. There is also another way.

In the Program menu, you can start up UUhistle's 'interactive coding mode'. This mode allows you to mix code writing and execution in much the same way as a typical Python interpreter's interactive mode does. Each input is executed immediately. UUhistle's interactive coding mode adds three things to a typical interactive interpreter session:

- UUhistle shows a detailed visualization of how each input is executed;
- UUhistle conveniently displays all earlier inputs to make it easier to keep track of how the current state was reached
- UUhistle allows you to step back and forth in the session history (undoing and redoing inputs) via the control buttons.

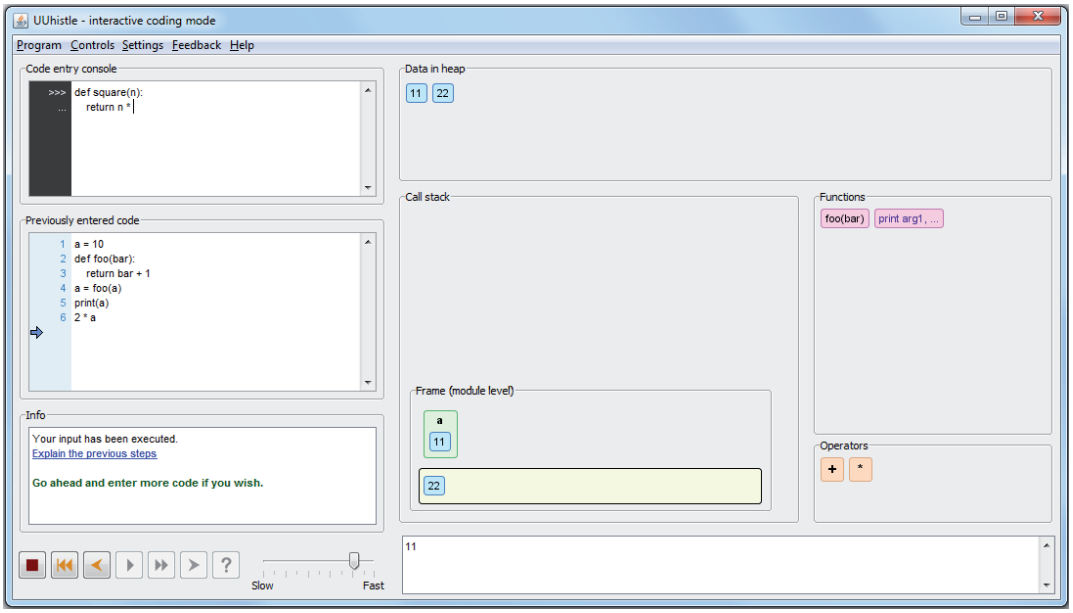


Figure 13.7: UUhistle's interactive coding mode. The user enters code at the top left. The new input appears below the earlier ones, and is immediately executed visually. Here, a few inputs have been executed and the user is typing in another function definition. Program output appears in the I/O console as normal. The value of the most recently evaluated input is visible in the expression evaluation area; here, `2 * a` has evaluated to 22.

The interactive coding mode is depicted in Figure 13.7.

After you type in a Python instruction and hit the Enter key, UUhistle starts animating the execution of the input right away, and shows all of its execution steps continuously. The GUI stays responsive during the animation, so you can keep typing in more code while the animation is running; the execution of each later instruction will be animated as soon as the previous animation finishes.

In this mode, pressing Undo cancels the execution of an entire previously entered instruction.

The interactive coding mode is intended for use both by learners for experimentation and by teachers for demonstration.

13.3 Teachers can configure program examples in many ways

One way for a teacher to provide an example program for students to view in UUhistle is to distribute the program code as a plain text file. Students can then store the file in their local file system and open it using UUhistle's Program menu.

Another way of distributing examples to students is to use a *course config file*. A course config file is an XML file in which the teacher can store a selection of Python programs and specify how they wish UUhistle to present those programs to students. Once the teacher has placed a course config file online – or distributed it in some other way – students can use UUhistle as a client to load the examples in it (by choosing Open Example Program in the Program menu). UUhistle can load a course config file either from a local path or a remote URL; the location of the file can be specified in UUhistle's general settings.

The UUhistle website `http://www.uuhistle.org/` gives a full list of supported configuration options and a description of the XML format. Below, I have listed some of the main things teachers can affect through a course config file.

- *Program descriptions*. Texts that highlight interesting points, provide example-specific advice to the learner, or give an overall idea of the example code. An example of how UUhistle displays the

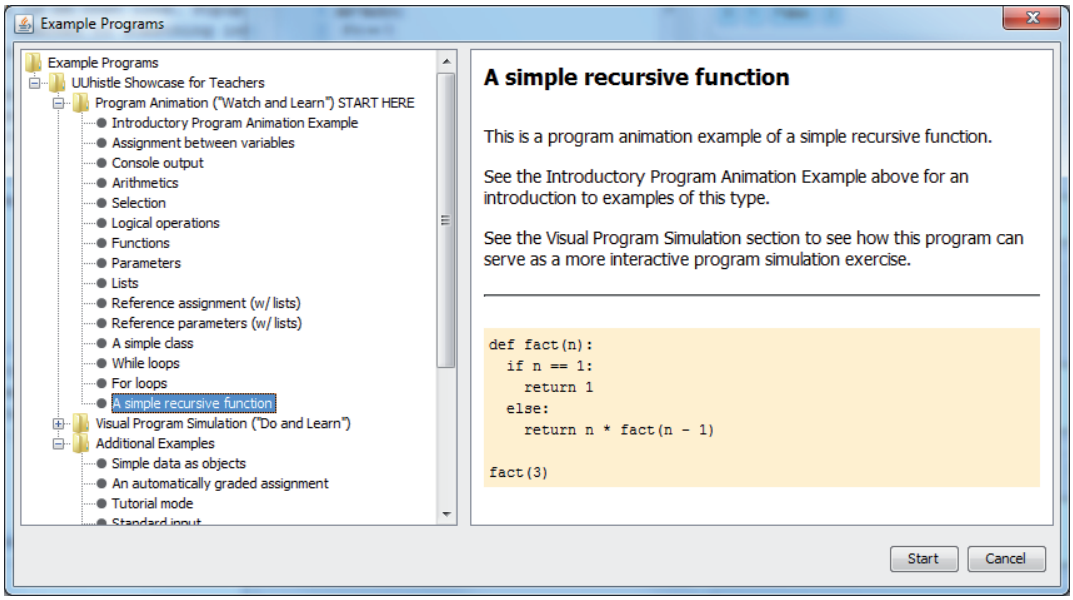


Figure 13.8: A selection of predefined program examples in UUhistle. This is the dialog that appears when the user chooses to open an example program through the Program menu. The user can select one of the programs defined in a course config file. If the teacher has provided a description of the example, UUhistle displays it on the right while the example is selected. Clicking Start loads the program into UUhistle's main window.

program description is shown in Figure 13.8.

- *Execution-time commentary*. Customized descriptions of individual lines of code that the learner can access through Info box links while control is on that line. Popup dialogs that appear at specific lines to alert the learner to interesting phenomena.

- *Grouping of examples*. Examples can be divided into named units, e.g., to correspond to stages of a programming course.

- *Visualization settings*. For instance, the teacher may wish an example to highlight how even simple values are objects.

- *Hidden code*. An example can include code which is not shown onscreen and whose internal behavior is not visualized. For instance, a 'hidden' class or function can be included that the visualized non-hidden example code makes use of as a black box.

- *Stop-and-think questions*. Dialogs that pop up when control reaches a particular line of code, and require input from the user. UUhistle supports multiple-choice questions and questions that can be answered with a single line of text.

- *Other settings*. Preset breakpoints. Disabling the Next Line button to force the learner to pause after each small step. Etc.

The teacher may configure settings at different levels of granularity: for a specific example program, for a unit of related programs, or for all the contents of a course config file at once.

A course config file can also be used to create visual program simulation exercises.

13.4 Teachers can turn examples into visual program simulation exercises

As described in the previous chapter, a visual program simulation exercise puts the learner in the driving seat. Instead of watching an animation, the learner has to read the given code and manipulate UUhistle's visualization to execute the instructions, allocating and using memory to keep track of program state.

The learner can, through the Settings menu, enable visual program simulation for any program in UUhistle. However, a more likely scenario is that the teacher first designs a selection of example programs and places them in a course config file. In the config file, the teacher can then enable visual program simulation for some or all of those programs (while disallowing the use of the Next Step and Next Line buttons to simply view them). This turns the example programs into visual program simulation exercises.

How does a VPS exercise work in UUhistle?

13.4.1 To simulate a program, the learner directly manipulates the visualization

UUhistle provides the graphical elements that the learner directly manipulates to indicate what happens during execution, and where, and when. Any execution step that UUhistle can display when animating a program can also serve as a part of a VPS exercise: the learner can create variables and objects in memory, evaluate expressions, assign values, manipulate the call stack, pass parameters, and so forth.

An introductory VPS example

Let us again consider the Python statement `b = a + 1`, whose animation steps I listed on page 195 and illustrated in Figure 13.2. Here is a corresponding list of simulation steps that the learner needs to perform upon encountering this line of code in a VPS exercise.

1. Evaluate the subexpression `a` by dragging and dropping the variable's value into the expression evaluation area in the topmost frame.
2. Access the definition of the + operator by dragging and dropping it from the Operators panel to the expression evaluation area.
3. Evaluate the literal `1` by dragging and dropping it from the heap to the expression evaluation area.
4. Apply the sum operator by clicking on it. The result appears in the evaluation area automatically, replacing the arithmetical expression.

5a. (If the variable `b` exists already.) Assign the sum to `b` by dragging it from the evaluation area into the variable. The previous value is automatically overwritten.

5b. (If the variable `b` does not yet exist.) Create the variable `b` by assigning the sum to it: drag the sum from the evaluation area to the area reserved for variables in the active frame (once you grab the value, an outline appears in the frame to highlight the possibility to drag the value there). Upon dropping the value in the new variable, name the variable through the context menu that pops up. The menu lists various identifiers that appear in the code from which to choose the name. (This process is illustrated using a different example in Figure 13.9.)

After these steps, control automatically moves to the next line.

GUI rules of thumb

In general, the GUI operations needed to perform simulation steps are highly analogous to the animations that UUhistle shows when viewing a program. Only a few different kinds of GUI interactions are used. Nearly all the simulation steps fall into one of the following four general categories.

- To access the contents of a memory location, drag the appropriate element with the mouse and drop it at the desired destination. The destination is either a variable (when assigning a value) or in the evaluation area (when forming expressions). This works on values in the heap and in the stack (including references), as well as on function and method calls and operators.

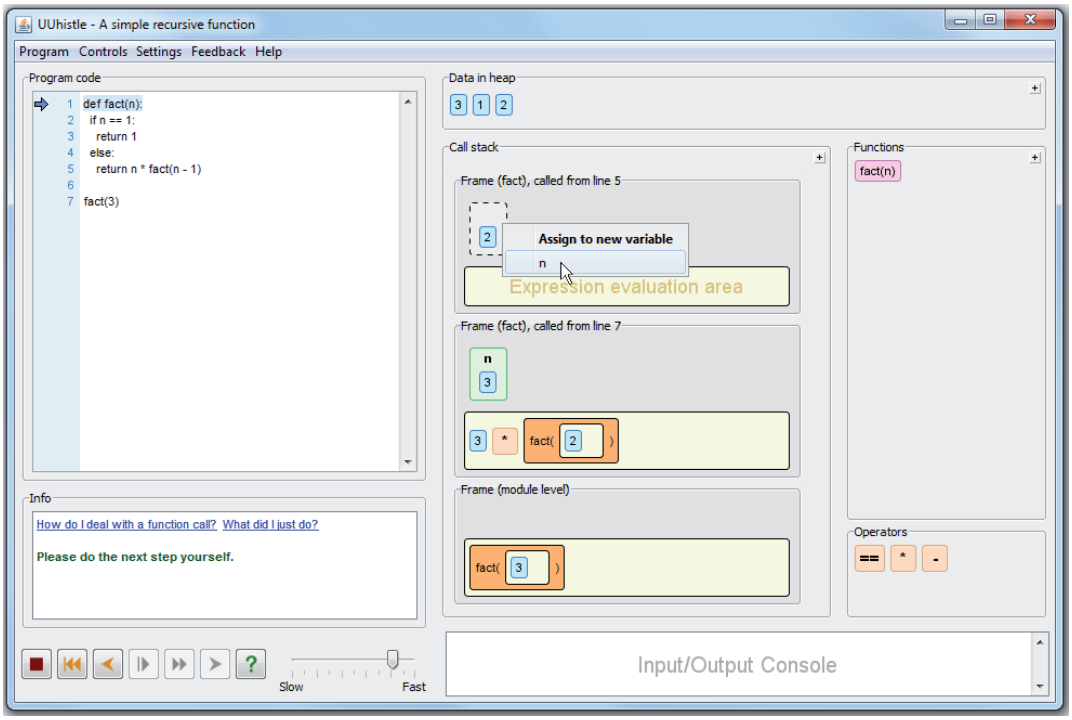


Figure 13.9: A visual program simulation exercise in UUhistle. The user is simulating the execution of a small recursive program, and has just dragged a parameter value into the topmost frame in order to create a new variable there. He is just about to name the variable using the context menu that has popped up.

- To invoke a routine, click on the appropriate element (operator or function/method call) in the evaluation area.
- To create a new element in memory (a frame, a function definition, etc.), click on the appropriate memory panel. This brings up a context menu in which you get to select what to create. To bring up the menu, you can click anywhere within the frame background or on the little plus in the upper right-hand corner, which is there only to hint that the frame is clickable. Exception: variables are created by dragging values, as described above.
- To move to a new line, click the line or its number. (This is not always required on default settings; see below).

More examples of simulation steps

I will now briefly describe how to perform certain common simulation steps in UUhistle.

To deal with `if` and `while`, first evaluate the condition expression. Once you have `True` or `False` in the evaluation area, click the line that execution proceeds to. On default settings, UUhistle's VPS exercises mostly do not require the user to manually control the 'instruction pointer'. With the exception of branching instructions such as `if`, UUhistle changes lines automatically as soon as the user is done with the current line. This is one of many default behaviors that can be affected by an associated setting.

To evaluate `a * b / 2 + (c + 1) * d`, first drag and drop the value of `a` into the topmost expression evaluation area, then do the same for the `*` operator and the value of `b`. Evaluate the subexpression `a * b` by clicking on the operator; the product appears in the evaluation area, replacing the subexpression. Add the `/` operator, then the literal `2` from the heap. Apply the operator. Add the `+`. Add the value of `c`.

Add another +, then the literal 1. Now evaluate the subexpression c + 1. Add a value of c, and click on the operator. Add a * and the value of d, and apply the multiplication operator. You are left with two numbers separated by a +. Finish up by clicking on the plus. This is the only step sequence for evaluating the expression that UUhistle accepts without complaint. You must make sure to evaluate subexpressions as soon as possible, but no sooner. This example illustrates how UUhistle leaves to the user the task of keeping track of the order of expression evaluation. Parentheses are never explicitly manipulated in UUhistle's VPS exercises or animations.

Function calls involve a number of steps. First, access the function definition: drag and drop the appropriate element from the Functions panel into the expression evaluation area. UUhistle automatically provides 'slots' (expression evaluation areas) where you can evaluate function parameters – that is the next step. Click the function call to signal the start of function execution. What happens next depends on whether the function is a built-in or defined in the code that is to be simulated. In the case of a built-in function (including 'hidden code'; see Section 13.3 above), the function is immediately executed, and a return value produced that replaces the function call in the evaluation area. Otherwise, you need to deal with the function implementation in more detail, as follows. Create a stack frame by clicking on the stack and choosing the appropriate option (New frame) from the context menu. Then pass the parameters: drag each parameter value from the call into a new variable in the top frame. After all parameters are in, execute the function body. The function call finishes with a return command: evaluate the return expression and drop the result onto the calling expression in the frame below. The top frame automatically vanishes from view.

To create an object, first click on the heap and select the appropriate type in the context menu. Then create a reference by dragging the object to the evaluation area. Initialize the object by invoking an __init__ method (a constructor-like method in Python) on it, if it has one. Calling __init__ or any other method works just like calling a function except that you need a reference to the target object in the evaluation area before you drag the method there.

Shortcuts

Assignment always involves evaluation of the right-hand side. Evaluation in UUhistle takes place in the expression evaluation area. For convenience, however, UUhistle accepts 'shortcuts' in trivial cases such as a = b or a = 10: you can drag the value of b or the value 10 directly into the target variable without first placing it in the evaluation area.

Similarly, a trivial return statement such as return True or return my_var can be executed as a single step by dragging the return value directly onto the calling expression in the second-from-top frame. The 'proper way' of first placing the return value in the evaluation area of the top frame also works.

13.4.2 UUhistle gives immediate feedback

In a VPS exercise, the learner needs to pick the correct execution steps from among many alternatives. There is plenty of room for making mistakes, as there should be, lest the VPS activity degenerate into mere controlled viewing with an unwieldy user interface.

Learners may misunderstand the semantics of many programming constructs and pick the wrong kind of execution step. They may use the wrong values or the wrong operators. They may not realize the need to allocate memory for frames or objects. They may perform steps in the wrong order. Rather than preventing mistakes like these, UUhistle allows the learner to make them and then gives feedback to help the learner fix what is wrong.

Whenever the learner makes a mistake, a warning dialog pops up, accompanied by a change of colors in the GUI; see Figure 13.10. The learner can disable the dialog as they become familiar with the system.

UUhistle encourages the learner to undo the incorrect step and try to find the correct solution. The Info box mentions the type of error and links to further materials.

Figure 13.10 shows a couple of example links. Asking "What was wrong with that?" brings up a more detailed explanation of what went wrong. The "What did I just do?" link invites reflection. It is possible for the learner to perform simulation steps whose meaning they do not themselves understand; clicking

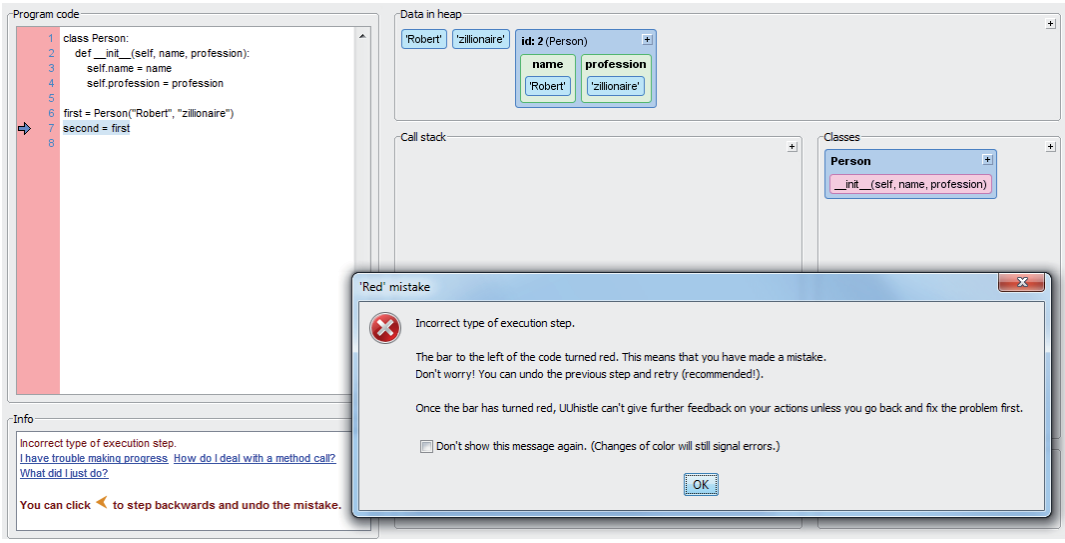


Figure 13.10: UUhistle reacts to a user mistake during a VPS exercise. A generic explanation of the problem appears in the Info box, with links leading to further reading. UUhistle uses the color red as an alert for errors where the user has performed an entirely wrong sort of execution step. When there is a 'lesser' problem with the step – e.g., the user assigns the wrong value to the right variable – yellow is used instead.

on the link summons an explanation of the previous (possibly incorrect) step, much like the "Explain the previous step" link in animation mode (cf. Figure 13.3).

UUhistle gives highly specialized feedback in some situations – for certain particular missteps performed in certain particular conditions. These specialized feedback texts address specific programming misconceptions that the student's mistake may reflect and suspected specific misunderstandings concerning UUhistle. Figures 13.11 and 13.12 show two examples of such feedback, which is accessible through the "What was wrong with that?" links. Section 15.3 says more about the way in which UUhistle is designed to address misconceptions.

In part because of the instant feedback, UUhistle's VPS exercises are not invulnerable to guessing and naïve trial-and-error strategies. These approaches are inconvenient because of the fairly large number of options from which the next step is to be picked, but the persistent student may still rely on them. When the user has trouble making progress – perhaps because of a naïve strategy – UUhistle attempts to encourage a more reflective approach, as shown in Figure 13.13.[3]

13.4.3 Many varieties of simulation exercise are supported

I have described above a basic sort of VPS exercise supported by UUhistle. Through a course config file (see Section 13.3 above), teachers can set up different kinds of visual program simulation exercises for their students.

In a vanilla exercise, the user needs to simulate each execution step, with the exception of line changes that do not involve explicit boolean conditionals (`ifs` or `whiles`). For a more thorough simulation, the teacher may require the learner always to change lines manually. This makes for more laborious exercises but immerses the learner in control decisions. It is necessary for the learner not only to know which line to jump to, but also to figure out exactly when the previous line is done with.

The teacher may also set up programs as animation/simulation hybrids. In a hybrid exercise, the learner can view parts of the program using Next Step and Next Line (or Run until line), but has to simulate

[3]This is one of the features added to recent versions of UUhistle that are partially inspired by the empirical work that I present in Part V.

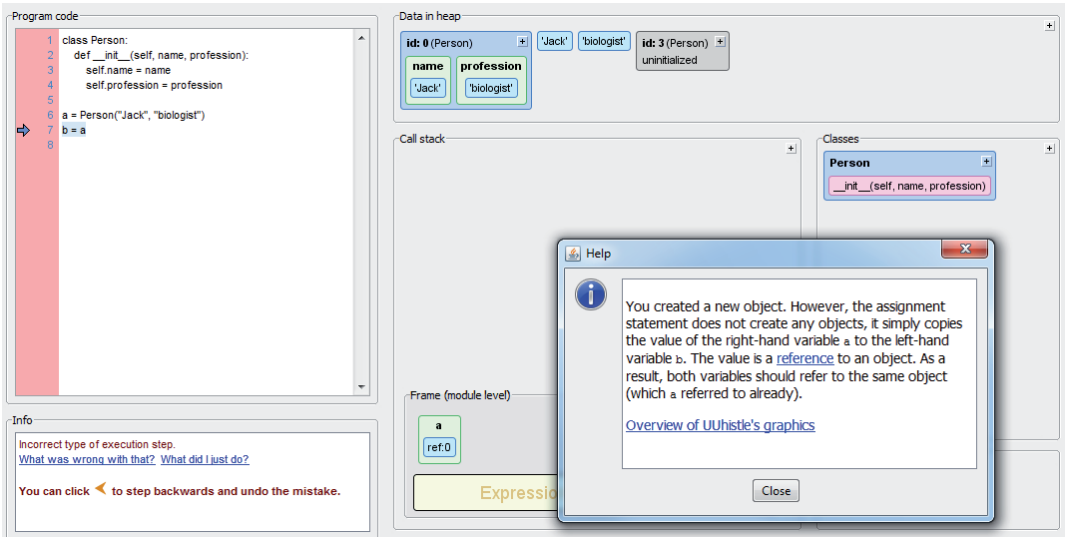


Figure 13.11: The user has created a new object when they should have merely formed another reference to an existing object. UUhistle's feedback – which the user accesses by asking "What was wrong with that?" – seeks to address a suspected misconception concerning object assignment.

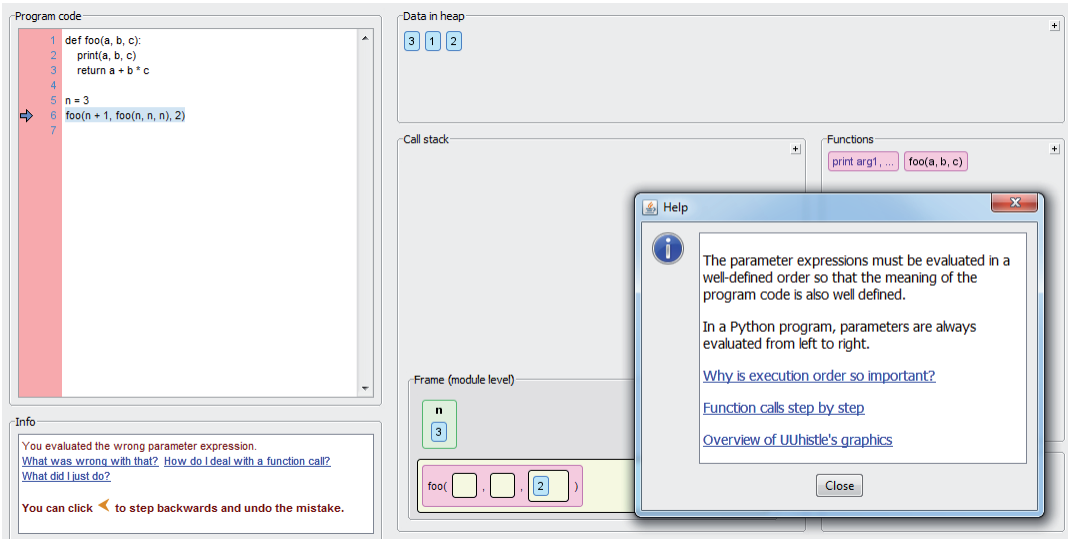


Figure 13.12: The user is evaluating parameters in non-standard order. They have possibly thought that the order does not matter and they might as well start with the simplest parameter expression.

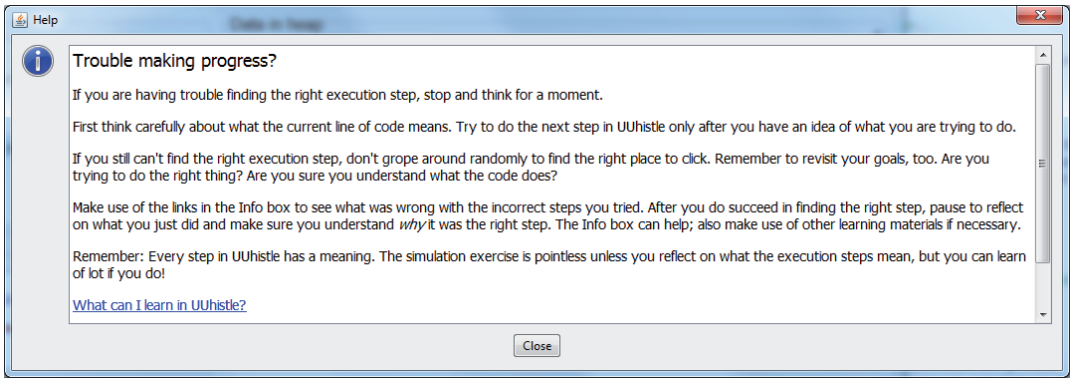


Figure 13.13: UUhistle attempts to encourage reflection and discourage guessing. The "I have trouble making progress" link, which brings up this dialog, appears in the Info box if the user makes a sequence of two mistakes: misstep→undo→misstep.

key points manually. In the config file, the teacher can specify either that certain types of simulation step must be simulated while others are animated (e.g., basic arithmetic is animated but function calls are not), or that only particular lines of the program code must be simulated. For a line that is executed multiple times, the teacher may specify that it must be executed manually a set number of times (e.g., one) but that later iterations are automatic.

It is possible to allow animation and visual program simulation simultaneously for a program example. In this case, the user can choose at each step whether to view it or simulate it. This is probably mostly useful for teachers experimenting with UUhistle.

The teacher may enable a tutorial mode (Figure 13.14) in which the Info box guides the learner at each step. Even if the tutorial mode is disabled, the teacher may allow the learner to ask for similar 'hints' by clicking on the Hint button (the one with the question mark). A maximum number of hints allowed per exercise can be set.

Just as when configuring program animation examples, the teacher can affect a VPS exercise's visualization settings, 'hidden code', stop-and-think questions, etc., as described in Section 13.3.

Again, for more details of the configuration options, see the web site `http://www.uuhistle.org/`.

13.5 UUhistle can automatically grade solutions

As UUhistle is capable of checking the correctness of answers, it can also do automatic grading. Teachers can define grading policies for VPS exercises in the course configuration file. A point value can be set for each exercise. The learner may be given a partial score for each correct step or may be required to complete the entire exercise correctly to gain any points. Control freaks can fine-tune the relative values of code lines. The effects of tutorial mode and hints (if enabled) on grading can be specified.

Stop-and-think questions (Section 13.3) can also be given point values. This can be used to award points for viewing non-VPS animations.[4]

When the end of a graded example is reached, UUhistle asks the learner whether they wish to submit their solution online. UUhistle can send solutions and grades as per the hypertext transfer protocol (HTTP POST). This facilitates integrating UUhistle with a web-based course management platform, for instance. Details on this process can be found at the UUhistle web site.

[4]Some teachers might wish to give point values for program animations with no multiple-choice questions, as a carrot for bothering to load up and view the examples; UUhistle supports this, too.

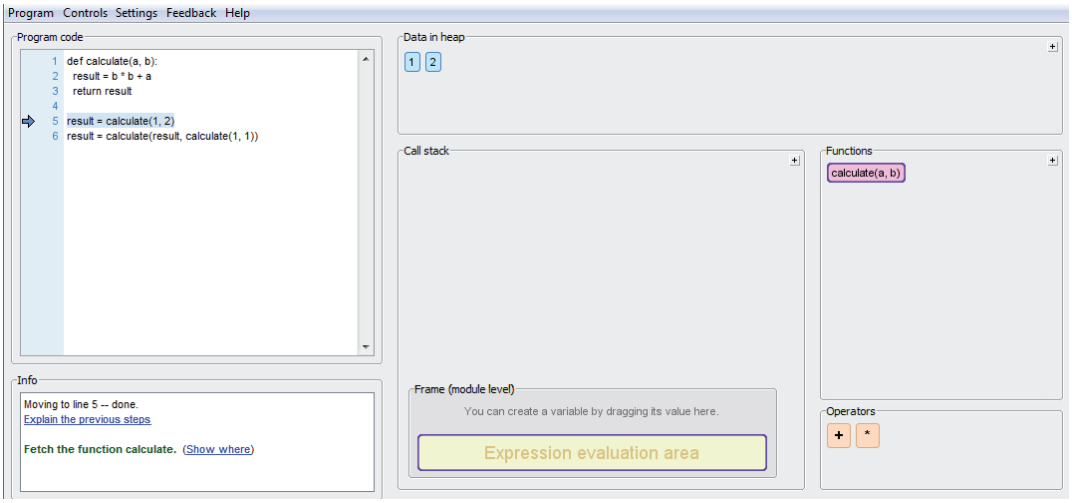


Figure 13.14: A VPS assignment in tutorial mode. The Info box guides the user. Clicking on "Show where" highlights the part(s) of the visualization that the next step applies to. For instance, here the function definition and the evaluation area are highlighted with a thicker, violet border and an animated, flickering background color.

13.6 UUhistle is built to engage students with visualizations

UUhistle supports multiple ways of engaging with a visualization. Figure 13.15 illustrates how UUhistle fits into the 2DET engagement taxonomy from Section 11.2.3.

Program animation in UUhistle, on default settings, is controlled viewing. It can turn into mere viewing if the learner only plays the animation continuously instead of advancing step by step. Just watching someone else work in UUhistle is also viewing, of course. Answering stop-and-think questions is a form of responding.

Visual program simulation in UUhistle involves applying given visual elements to a program tracing task.

In theory, learners could do visual program simulation on programs that they wrote themselves. Technically, UUhistle supports this. In practice, however, VPS may be best used as a tool for instructional design so that teachers create VPS exercises for students to solve (see Chapter 14). In terms of the 2DET, VPS is likely to work best with given content. The same goes for stop-and-think questions on the responding level.

Working in the interactive coding mode (Section 13.2) is a variety of controlled viewing of own content.

More or less any program visualization can be presented to others. UUhistle's visualizations are no exception.[5]

13.7 The system is available at UUhistle.org

UUhistle v0.6, the newest release at the time of writing, is available free for non-commercial use at `http://www.uuhistle.org`.

UUhistle is written in the Java programming language. It can be run as an applet in a web browser or downloaded as a standalone application.

There are two different builds of UUhistle available at the web site. The slightly simpler 'lean build' is essentially a client for ready-made example programs – animations and VPS exercises – that have been

[5] UUhistle provides little direct support for learners to present or share their visualizations. It does feature a markup pen that can be used to draw on top of the visualization while the right mouse button is kept pressed. This may improve the visualization's 'referencability' (Hundhausen, 2005).

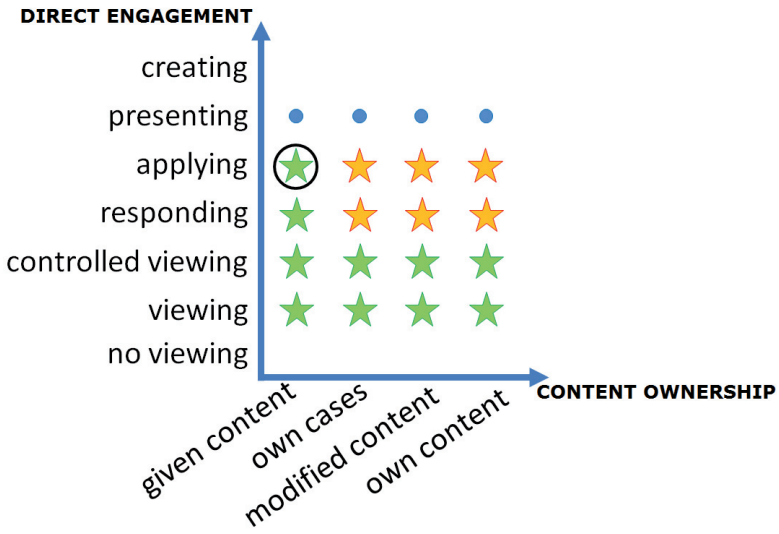


Figure 13.15: UUhistle in the 2DET engagement taxonomy. Stars mark the forms of engagement that UUhistle supports. However, the forms of engagement that correspond to the orange stars in UUhistle are perhaps not sensible in practice. The circled star corresponds to the recommended form of visual program simulation: learners applying given visualizations to teacher-created programs. UUhistle provides little specific support for the presenting level of direct engagement, but its visualizations can, of course, be presented to others.

stored in course config files. The full build additionally allows the user to edit program code, and open and save source code files locally.

13.8 UUhistle has a few close relatives

There are other educational software visualization systems whose functionality overlaps with UUhistle's. Many systems were reviewed in Chapter 11; let us compare and contrast UUhistle with some of its closest relatives.

13.8.1 UUhistle is the love-child of Jeliot 3 and TRAKLA2

Jeliot 3, described on page 165 above, uses the same level of abstraction and similar graphics as UUhistle to visualize Java programs. When it comes to animating program execution as graphics, UUhistle is by and large a Python equivalent of Jeliot 3. Jeliot does not support visual program simulation.

TRAKLA2 (Korhonen, 2003; Korhonen et al., 2009a) is an algorithm visualization system developed within the same research group as UUhistle. Its standout feature is its support for automatically assessable *visual algorithm simulation* assignments: the student starts out with a description of an algorithm and a visualization of a related data structure, and uses GUI operations to directly manipulate the visualization. The goal is to demonstrate and learn about how the given algorithm works. For instance, an exercise might present the student with a pseudocode algorithm for adding values to an AVL tree, and the student uses the GUI to show where and when nodes are added and rotated. As is typical with an AV system, the level of abstraction in TRAKLA's visualizations and simulations is high (Figure 13.16).

UUhistle combines the lower-level abstractions of Jeliot with the engaging mode of interaction in TRAKLA2. One might say that UUhistle is to program visualization what TRAKLA2 is to algorithm visualization.

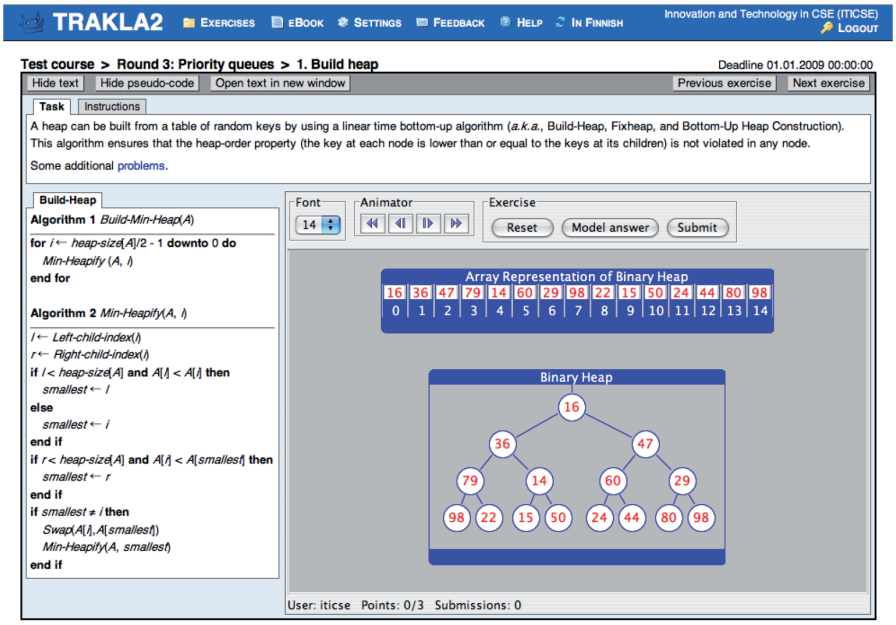


Figure 13.16: A visual algorithm simulation assignment in the TRAKLA2 system (image from Korhonen et al., 2009a).

13.8.2 A few other systems feature support for visual program simulation

The Online Tutoring System (p. 181) and Dönmez and İnceoğlu's visualization system (p. 179) each support a limited form of visual program simulation. However, UUhistle differs from these tools in a number of significant ways. UUhistle's scope is wide: it supports a larger subset of a programming language, including support for functions, user-defined classes, references, and recursion, among other things. UUhistle also explicitly emphasizes the various uses of computer memory during execution, and makes the call stack central to program state.

Another difference is in the use of graphics: UUhistle's visualization is a graphical abstraction of memory, while the Online Tutoring System was largely based on text, and Dönmez and İnceoğlu's tool makes heavy use of widgets such as dialogs, radio buttons, and drop-down lists, which the student uses to select what the computer does next. Arguably, UUhistle's user interface is less cumbersome than the those of the other two tools, which require the user to repeatedly type in text in order to name identifiers or specify the values of expressions. (We will have more to say about the details of UUhistle's design in Chapter 15.) The breadth of UUhistle's feature set and configurability also appears to significantly exceed those of the other two tools. Each of the three systems visualizes programs written in a different language. Unlike the other two tools, UUhistle supports different modes of visualization use within the same system.

The ViLLE system also features a form of VPS assignment, which is currently at an early stage of development (see p. 181).

Gilligan's PV system (p. 177) was based a similar form of user interaction to UUhistle's VPS assignments, and shared with UUhistle the goal of teaching novices about program execution. However, Gilligan's programming-by-demonstration approach was crucially different from visual program simulation in that the simulations were used to create programs rather than trace the execution of existing ones.

Some other systems feature user-controlled simulations at different levels of abstraction. JV^{2}M (p. 183), for instance, supports a form of VPS at a bytecode level. ViRPlay3D2 (p. 177), on the other hand, works at a very high object-interaction level. Both systems are based on three-dimensional virtual-reality worlds, which sets them further apart from UUhistle.

The intelligent tutoring system *LISP Tutor* (see, e.g., Anderson et al., 1989) featured a mode of interaction similar to VPS, albeit without visualization: the learner was expected to input the stages of expression evaluation of LISP programs.

Now that we have presented a definition of visual program simulation, and a supporting software system, it is time to link VPS to learning theory.

Chapter 14

Visual Program Simulation Makes Sense

In this chapter, I draw on the literature presented in Parts I to III to analyze visual program simulation and explain why it is a sensible pedagogical approach in the light of learning theory and empirical research.

Throughout this chapter, I often refer to previous chapters and sections rather than directly to the original literature; the reader will find literature references in those chapters and sections.

I have structured this chapter as follows. Section 14.1 establishes that VPS addresses important educational concerns. Section 14.2 considers the importance of reading code in CS1 and how VPS serves as a vehicle for learning to read code. In Section 14.3, I discuss the form of active learning that VPS promotes and how it fits in with the software visualization literature. Section 14.4 explains why and how a supporting software system is useful, if not critical to the success of VPS. Finally, Section 14.5 points out a few of the main weaknesses of visual program simulation.

14.1 The literature widely highlights the importance of program dynamics

Several strands of computing education research have suggested in different ways that a crucial challenge in introductory programming is learning to reason about the behavior of program code as it runs. It is this challenge that visual program simulation seeks to address.

14.1.1 VPS targets improved mental representations of program dynamics

In the psychology of programming literature, the challenge of program dynamics is sometimes phrased in terms of a notional machine – an abstraction of a computer as the executor of programs of a particular kind (Section 5.4). A notional machine implements the runtime semantics of programming language constructs and takes care of the 'hidden' processes that make programs work. The importance of learning about a notional machine has gained widespread agreement and empirical support in the literature, as Chapter 5 showed in some detail. Below, I summarize the arguments and relate them to visual program simulation.

Mental models of a notional machine

People form mental models of the systems that they interact with. For the novice programmer, one such system is the notional machine that the student learns to control as they learn to program (Chapter 5). A novice's initial mental model of a notional machine is likely to be – typically of mental models in general – incomplete, unscientific, deficient, lacking in firm boundaries, and liable to change at any time. It may be based on guesswork that draws on superficial program characteristics such as keywords and identifiers. Despite these shortcomings, the learner may feel comfortable with the model and rely on it while developing behavioral patterns for programming. Novices may also use multiple, possibly contradictory models to deal with different situations.

By contrast, experts' mental models are more stable and accurate, and draw on general principles rather than superficial characteristics. Learning should facilitate the evolution of students' models so that they have these features. Aiding mental model formation as early as possible is important, as

changing an ingrained but flawed mental model is more difficult than helping a model to be constructed in the first place. Mental model research further predicts that novice programmers can be expected to have trouble transferring their mental models of a notional machine to an even superficially different programming language unless the original model is well ingrained through a substantial amount of practice. A programming teacher can affect the formation of mental models via a conceptual model, that is, an explanatory device that is used to explicate the concepts behind the system.

In these terms, visual program simulation can be characterized as follows.

> Visual program simulation combines a conceptual model of a notional machine with a learning activity that teaches about the model. It aims to facilitate the formation of principled, stable, and consistent mental models of a notional machine. VPS is a form of practice that helps the student ingrain their mental model.

Misconceptions and difficulties with state

Many of novice programmers' specific misconceptions and limited understandings can be explained by a lack of a viable mental model of the notional machine (see Sections 3.4 and 5.4, and Appendix A). These misconceptions often have to do with 'hidden' runtime mechanisms which are implicit in the concrete code, such as parameter passing, constructors, and references. Perhaps the most significant are the more general misconceptions about the nature of the computer or the programming environment, such as attributing human-like cognitive capabilities to the computer.

The relationship of programming language elements to what happens during execution – underlying memory usage in particular – has emerged as a major theme in misconceptions research and a major educational challenge in introductory programming education. Psychological studies of program comprehension have highlighted program state – keeping track of which is a central responsibility of a notional machine – as the aspect of programs that is the most difficult to understand (compared to other aspects such as control and data flow).

> By making the notional machine explicit, visual program simulation aims to address students' misconceptions about computer capabilities. By explicating memory and the effects of code constructs on memory, visual program simulation aims to address students' misconceptions about those constructs and to help students reason about program state.

Programming schemas

One line of thinking within the psychology of programming literature is that the most important challenge in learning to program is to learn to 'put the pieces together'. In other words, the challenge is to learn and apply higher-level schemas that suggest how to combine the relatively simple code instructions so that they solve a particular kind of problem (see esp. Sections 4.4 and 5.6). These higher-level schemas (e.g., how to compute an average) build on lower-level ones that correspond to notional machine primitives (e.g., how to assign a value to a variable or how to send a message to an object) and can themselves be used as building blocks for ever more abstracted schemas.

Studies suggest that both novices and experts use a mix of top-down and bottom-up strategies when writing and reading programs. Applying known schemas at a higher level allows the programmer to adopt a more efficient top-down approach. Choice of strategy is determined by the availability of schemas in the programmer's repertoire: people use top-down strategies when possible, and resort to bottom-up approaches when faced with difficult or novel problems for which they lack existing schemas. The complete beginner does not yet have any schemas and will have to work on programs bottom-up, reasoning from

the individual instructions of the programming language upwards, or try to apply a higher-level template despite not understanding its 'pieces'. The automation of low-level schemas is necessary for the successful formation and application of complex high-level schemas.

> Visual program simulation encourages the formation of lower-level schemas that correspond to the constructs of the programming language and the behavior of the notional machine. The lower-level schemas are the basis for understanding and applying problem-solving patterns at higher levels of abstraction.

Tracing and debugging

Students have trouble with the key programming skill of tracing (Sections 3.3 and 5.5). Tracing is needed when a schema-based top-down approach to program comprehension fails as a result of incorrect expectations or bugs in the program. As novices do not have many schemas and their programs are buggy, they need tracing even more often than experts do. In addition to being useful during program comprehension and many debugging tasks, tracing is also an important part of the program design process.

In terms of mental model theory, tracing a program is the act of 'running' a mental model of the program and the notional machine that executes it. Tracing a program that is behaving unexpectedly or is of an unfamiliar variety requires a mental model that is robust, that is, based on general properties of each program component rather than a specific configuration of code elements seen in a previously encountered example. Experts tend to have robust mental models, whereas novices do not. Robust mental models can be fostered by separating programming constructs from the functions they serve in particular programs, by inspecting the constructs in isolation, and by using them in different construct–program configurations.

Semantics are defined for language primitives rather than their combinations. A notional machine running a program is only concerned with those primitives, not with what surrounds it. VPS promotes the ability to take a similar focus when needed. The student is to focus on each instruction to be simulated locally, without caring – for the moment – about its function in the program. The correct action is determined directly by the particular piece of code currently under consideration. This is in line with the 'esthetic principles' for robust models put forward by de Kleer and Brown (Section 5.2).

> Visual program simulation is a way of practicing the stepwise tracing of programs, a skill that is intended to be transferred out of the VPS environment and into authentic programming situations. Especially if the program examples are designed in such a way that they allow the learner to examine programming constructs in different combinations, VPS examples can aid the formation of robust mental models that are useful in debugging.

Tracing a program involves keeping track of the current state of the program run in terms of the elements of a notional machine. To manage this the limitations of working memory, the programmer needs to successfully select the 'moving parts' that they keep track of and a suitable level (or levels) of abstraction. Expert programmers also often make use of external representations (e.g., the view of a debugger) to help them to avoid cognitive overload. Novices are less adept at choosing what to keep track of and often fail to make use of external representations. Visual program simulation may help here, much as many other forms of program visualization can:

The visualizations used in visual program simulation can serve as external status representations that help the novice manage cognitive load while tracing a program. The visualization may suggest to the learner what kinds of 'moving parts' it is useful to keep track of while thinking about program execution. Using a visual program simulation system might serve as a basic platform for learning to make use of external status representations that are used by professionals (i.e., debuggers).

Program and domain models

Many studies of novice and expert program comprehension have been conducted, with a mix of results (Section 4.6). One of the clearest areas of agreement is that when interpreting an actual program that accomplishes a task, a successful comprehender forms models both of the program and of the problem domain, and relates the two models. Teachers should therefore look for ways to foster the creation of each kind of model. Where does VPS fit into this scheme of things?

Visual program simulation supports the formation of program models during program comprehension, including the static and dynamic aspects of programs and their relationship. VPS does not directly address the formation of domain models or the linking of domain models to program models – such learning has to be facilitated in other ways.

Teachers can certainly try to design and present visual program simulation examples in a way that encourages the formation of domain models, but the simulation activity does not contribute directly to domain knowledge as the perspective is that of a notional machine.

Studies cited in Section 4.6 have suggested that experts form both domain and program models, while novices tend to focus on either the program model (perhaps especially when taught in the procedural paradigm) or on the domain model (perhaps especially when taught in the object-oriented paradigm). A conjecture from these results is that object-oriented novices in particular need help with the creation of program models as their paradigm emphasizes the problem domain. If this is the case, VPS is potentially of more assistance when used in an objects-early course or in some other form of CS1 that emphasizes domain modeling.

14.1.2 VPS seeks to help students construct knowledge below the code level

Chapter 6 introduced several variants of constructivist learning theory. Visual program simulation is a poor fit with those more extreme forms of constructivism that denounce ontology and educational norms. However, it sits better with a more conservative constructivism that emphasizes the need to facilitate the construction of viable knowledge that enables one to interact successfully with the world.

Ben-Ari's interpretation of cognitivistically tinged personal constructivism (Section 6.7) gives a few pertinent points for the programming teacher to consider. First, beginner programmers lack a model of the computer that is effective for the purpose of programming. Second, students inevitably construct their own understandings of what goes on beneath the level of abstraction that one teaches at, which in CS1 is typically a level that corresponds to program code. And third, the computer forms an accessible ontological reality and is a merciless, unnegotiating judge of the correctness of students' knowledge. Taken together, these points imply that students should be helped as early as possible to construct viable understandings of the abstractions underlying program code, and guided to improve on any non-viable understandings that they harbor as they enter CS1 or that they construct during their programming studies.

Visual program simulation seeks to help the novice programmer construct viable knowledge on a level of abstraction that underpins the code level that students usually operate at. Part of this process is coming to see the non-viability of one's existing knowledge. Incompatibly with some forms of constructivism, visual program simulation relies on the assumption that there are correct answers which are defined by the technical reality of the computer.

14.1.3 VPS seeks to help students experience dynamic aspects of programming

The phenomenographic tradition of educational research (Chapter 7) sees learning primarily as qualitative changes in the relationship between the learner and aspects of his experiential lifeworld. The most significant moments of learning are those in which the learner gains the ability to perceive a phenomenon in a different, richer way than before. Phenomenographic learning theory suggests that educationalists must find the specific aspects of the content of teaching that are critical to such qualitative changes in learning.

Phenomenographic work within CER (Section 7.5) has underlined a key early challenge in introductory programming education: novice programmers have trouble learning to think about programs not only statically in terms of program code, but as dynamic runtime entities. Analogous results have been discovered regarding programming in general, which is sometimes experienced merely as writing program text and not in terms of controlling execution, and also regarding specific programming concepts (object and class) that are sometimes perceived merely as pieces of code. The pedagogical solution to this problem, from the phenomenographic point of view, is to help learners become focally aware of the execution-time behaviors of programs and their relationship with instances of program code.

Visual program simulation seeks to create situations that encourage learners to focus on the execution-time behavior of programs as they are run by a computer. This enables learners to discern a dimension of variation corresponding to programs' dynamic behavior. Further, focusing on program code and behavior simultaneously allows learners to discern the relationship between these two aspects of programs.

14.1.4 VPS seeks to help students across the threshold of program dynamics

Like phenomenography, the theory of threshold concepts (Chapter 9) also emphasizes the importance of the content of learning for pedagogy. The theory maintains that within curricula there are particularly powerful – but troublesome – discipline-specific concepts or ideas that lead to new kinds of thinking and practicing and new opportunities for learning. These threshold concepts demand a focus on teaching and learning, even at the expense of other important concepts, as failing to master a threshold concept may make further learning difficult or even impossible.

Program dynamics is a plausible threshold concept in computer programming as it appears to have the main characteristics of such concepts. It is integrative, explaining relationships between other programming concepts. It is transformative, as mastering it leads to a dramatically different dynamic view of programs as 'more than just code' and makes possible a new kind of reasoning about programs through tracing. It is troublesome for many learners, as is well documented in the research literature, partially because it tends to be part of experts' tacit 'obvious' knowledge. It tends to be irreversible in the sense that learners do not forget it once they master it and have difficulty understanding how someone who is yet to cross the threshold thinks. It sits at a boundary between schools of thought within the discipline of computing. Finally, learning about program dynamics involves making a universal, everyday idea – state – central to discipline-specific thinking, a feature that also appears to characterize many threshold concepts.

Visual program simulation seeks to promote a teaching and learning focus on the candidate threshold concept of program dynamics. Crossing this threshold is a crucial early step in helping novices to think about and work with programs as programmers do. Failure to cross it is a serious barrier to further learning and may result in a significant waste of time and energy on the part of both learners and teachers.

14.2 Reading code contributes to other programming skills

Visual program simulation is an activity that is based on reading program code. Reading code is, of course, a useful professional skill in itself, but is also useful in education because of its impact on the development of other skills. I have presented the foundations for the arguments in this section in Chapters 2 and 3, and Sections 4.5 and 10.1.

14.2.1 VPS is a stepping stone towards writing code

Some recent work in CER has sought to establish the relationships between learning to trace, explain, and write program code (Chapter 3). If there exists a learning hierarchy in which the ability to trace code precedes the ability to explain comparable code, which, in turn, precedes the ability to write similar code, then it would make sense to start students off with code-reading assignments and progress from there towards writing. This could alleviate the problem that was highlighted by the educational taxonomies discussed in Chapter 2: the typical goal of CS1 education (writing original programs) is cognitively very demanding for the introductory level.

Visual program simulation directly targets the third level of (the revised) Bloom's taxonomy: students apply their knowledge of programming constructs and the notional machine to simulate program execution. This category has a cognitive complexity that is appropriate for introductory courses.

Clear evidence of a learning hierarchy has not emerged, and some studies have highlighted significant challenges in transferring tracing knowledge to code-writing situations. Nevertheless, recent studies suggest that even if there is no strict hierarchy, the skills of tracing, explaining, and writing can reinforce each other, and it makes sense to nurture their development in tandem.

Visual program simulation seeks to aid the development of code-reading skills both for its own sake, and for its supporting role in the development of code-authoring skill, which is the primary goal of programming education.

VPS assignments, like all program visualizations of notional machines, focus on exposing how programs work, not on the steps involved in creating programs. VPS trains students in tracing skills, which may be of more direct benefit in building debugging skill rather than program-writing skill. While tracing is not needed in all debugging situations, it is needed in many of them, and debugging is, of course, a necessary skill for any substantial code-authoring project.

14.2.2 VPS exercises can have elements of worked-out examples and unsolved problems

Let us again consider Table 10.1 on page 122, which gives a rough overview of different approaches to teaching CS1. Where does visual program simulation fit in?

There is some leeway in how visual program simulation can be used. VPS assignments can be done individually or in groups. They can be offered to students 'because they will be useful later' or can play a part in a more inquiry-driven curriculum (e.g., "How is it possible that this is what happens when this program runs?"). They can feature planlike programs with explicit goals and subgoals (although accentuating high-level goals is not a part of VPS itself) or unplanlike programs that students simulate to improve on their low-level schemas. They can perhaps be used to present contextualized examples as well as decontextualized ones (although this may be challenging in practice; see Section 14.5 below).

Overall, however, it is clear that visual program simulation is a form of direct instruction and matches many of the features listed on the right-hand side of Table 10.1. VPS exercises promote learning about fundamentals rather than direct practice of authentic professional skills. They use given programs rather than have students design their own. They involve reading rather than writing. They are small, closed, highly structured, and address teacher-set goals and content. They involve an artificial form of assessment.

> Visual program simulation is a tool for measured instructional design when teaching about fundamental programming concepts. It serves to lay a foundation for completing larger, more complex, and more authentic tasks that can be interleaved with VPS exercises.

Worked-out examples of writing code

Table 10.1 also contrasts having students solve problems and having them study worked-out examples, i.e., expositions of solutions. Worked-out examples can be used in instructional design as a device for managing cognitive load (Section 4.5) and thereby to foster schema formation and learning in general. A weakness of worked-out examples is that they may not engage learners to study them carefully enough. Interspersing worked-out examples with problem-solving tasks has been shown to be an useful pedagogical strategy.

A programming problem generally means writing a piece of code in order to do something – either something that is directly useful or something intended merely to serve an educational purpose. The simplest type of worked-out example for such a problem is an example program: the learner can study the way an expert has put together language constructs and the way the constructs combine to produce program behavior. To the extent that experts forward-develop the kinds of simple programs shown to novices, a piece of code also exposes something of the sequence of steps involved in creating the program.[1]

> A visual program simulation exercise can serve as a type of worked-out example of program writing, as it presents program code for the learner to study. In this way, it can serve to manage cognitive load and aid the formation of programming knowledge.

The manner in which a VPS exercise can serve as a worked-out example is defined (and restricted) by the particular VPS example and its documentation. An 'unplanlike' piece of code that does not address meaningful high-level goals is only an example of how a few constructs can be put together in order to

[1] Merely presenting the code and how it works is nevertheless only a limited sort of worked-out example. A more elaborate worked-out example would clearly explicate the expert's reasoning while writing the code, and dissect the program in terms of its subgoals and their solutions.

examine what they do.[2] A more 'planlike' program will also serve as an example of applying higher-level plans to solve a problem. The downside of planlike programs is that they tend to be lengthier and more laborsome to simulate visually.

Problematizing program animation

When it comes to the topics of program tracing and the behavior of a notional machine, a visual program simulation is not a worked-out example but an educationally motivated problem that demands to be solved. Program animations – that is, merely viewing execution – can be seen as worked-out examples that correspond to visual simulation problems.

The second nature of VPS assignments as tracing problems may mitigate a recognized weakness of worked-out examples, namely, that students sometimes fail to engage with the examples provided. An explanation of how the example program works is only gradually uncovered during a VPS exercise, and the learner has to work to access it. Hypothetically, the interaction that VPS assignments demand increases the average student's germane cognitive load compared to program animations, as VPS requires the student to anticipate future solution steps.

> Visual program simulation is a way of problematizing the execution of programs and the behavior of a notional machine on a level of abstraction lower than program code. It is a way of presenting students with execution-related problems for which more passive program animations can, in turn, serve as example solutions. The interactive nature of VPS is a way of encouraging and requiring students to pay attention to examples.

14.3 Visual program simulation makes active use of visualization

Many theories of learning have been used to support the use of program visualization, among them mental model theory, constructivism, and the phenomenographic theory of variation (Section 10.3). Program visualization also has a credible track record in CS1 education, and various studies have reported a positive impact on learning from using a program visualization system (Section 11.3). Reviews of software visualizations in education have generally concluded that visualization can be effective, but is not a panacea.

14.3.1 Visualizations work best with active engagement

Actively engaging learners with visualizations has been suggested as an important factor in the educational success of a visualization (Section 11.2). This is in line with advice from educational research and various learning theories that emphasize the role of active learning (see Part II).

What this means is that merely showing students a notional machine in action may not be enough, as this method fails to engage some learners enough for them to really study the visualization and make the notional machine a part of their own thinking. Further means of activating the learner are needed.

Applying a visualization

Researchers in CER have proposed taxonomies of increasingly engaging ways of interacting with a visualization; I proposed one such taxonomy myself in Section 11.2.

The simplest way to use a visualization is just to look at it, which may work if the learner is motivated enough to study the visualization in depth, but is likely to fail otherwise. A slightly more engaging activity is for the learner to control the viewing of a visualization in one way or another; this form of engagement

[2] This is also an authentic practice; experts write and study unplanlike programs when learning a new language, for instance.

features prominently in many program visualization systems which allow the learner to move through the execution of a program step by step at their own pace.

According to proposed taxonomies, responding to questions about a visualization is the next step up in engagement. This, however, depends greatly on the questions. It is possible for questions to point the learner's attention to important issues within the content of the visualization. A possible downside to posing questions is that they may direct the learner to focus narrowly on the specific issues that are asked about. Many questions about content may not in fact require the learner to actually understand the visualization. While understanding the visualization is not an end in itself, failing to engage with the visualization makes it unhelpful as a learning aid and does prevent the visualization from being used as a platform for thinking. Some common types of question are also vulnerable to guessing.

More engaging still than responding to questions is to manipulate visual components to carry out a contet-related task. This is where VPS fits in (see Section 13.6).

> Visual program simulation is a way of engaging with a visualization of program execution. It involves a form of direct engagement with a visualization that calls for a cognitive investment beyond that required for merely controlling execution or responding to occasional questions. Having the learner apply visual components to show how execution proceeds invites him to study the meaning of the visualization and to make the meaning part of his thinking. The requirement to interact with the visualization is likely to increase time on task compared to passive visualization; part of the hypothesized effect of VPS is due to the additional time spent on studying the example programs provided.

Manipulating the hidden

VPS requires the learner to manipulate what is normally hidden and automatic.

I observed in Section 9.4 that since threshold concepts tend to be abstract and tacit in expert knowledge, it is not a trivial matter to follow the advice of the theory's creators and engage students in manipulating the conceptual material. This, however, is precisely what visual program simulation intends to do with the possible threshold concept of program dynamics. Unlike in a programming exercise, in which learners work primarily at the code level, visual program simulation calls on the learner to manipulate what is usually not seen.

> Not only does visual program simulation make program dynamics – a perspective which is commonly automated by a computer and is often tacit and 'obvious' in expert knowledge – visible, it also invites students to manipulate this 'hidden' conceptual content both mentally and concretely. This follows advice from the threshold concepts research community.

An interesting parallel can be drawn between the way VPS requires the learner to simulate what can be – and usually is – automated and Wickens and Kessel's studies of process control training. To recapitulate from page 57 above, Wickens and Kessel found that people trained to manually control the behavior of a complex system formed better mental models of it than people who were trained to monitor the system as it was running automatically. The former group were able to transfer their learning to system monitoring and fault detection, while the latter group had trouble transferring theirs to manual control and performed worse on fault detection tasks. Similarly, an active hand in controlling programs may make a better notional machinist of the novice, even if such manual control will not be a part of their eventual routine as a programmer.

14.3.2 Using a given visualization has practical benefits

What about the highest levels – present, create – of the engagement taxonomies in Section 11.2? Why should every teacher not just have students present visualizations to others and create novel visualizations from scratch?

Presenting visualizations and creating novel ones is a viable way to learn about algorithms (see Hundhausen, 2002). The higher levels of the direct engagement could also be a part of visual program simulation. Learners could present how they simulate the execution of a program (although it is debatable if this would engage them any more than presenting a program animation). Instead of using a given visualization such as the one in UUhistle (on the applying level of the 2DET), a program simulation task could require them to design their own way of visualizing a notional machine (on the creating level) and using it to explain the execution steps of programs.

However, my discussion of VPS and the UUhistle system is primarily concerned with VPS realized as applying. The reason is that engagement is not all, and when it comes to program visualization in CS1, this level of engagement may be more feasible than the higher levels.

Designing a novel way to visualize program execution requires great cognitive involvement. Successful creation will doubtless be useful in leaning about program dynamics, but the risk of cognitive overload must be considered. Moreover, creating a program visualization may be next to impossible with fragile knowledge of the notional machine. Even using a ready-made external representation of execution is challenging enough if one has a vague or fragile understanding of the content, let alone creating a meaningful and useful representation oneself (Vainio and Sajaniemi, 2007). Fragile knowledge is very common in CS1. Asking students to create a novel, correct visualization of a notional machine would certainly be a way to *require* them to learn about program dynamics, but might not be the best way to *help* them learn.

It is also pertinent to wonder if creating novel program visualizations places too much of an emphasis on visualization design at the expense of other useful CS1 activities. The time investment is substantial even if learners do not need to go for a polished look. Having novices devise a generally viable way to visualize program execution is a big ask and will involve a long process. The presenting level suffers from some of these same weaknesses, albeit to a lesser extent, while engaging on the lower levels probably requires less time on average. A reasonable instructional strategy might be to use a judicious mix of engagement levels, as the creators of the original engagement taxonomy indeed suggested (Naps et al., 2003): some presenting here, maybe a bit of creating there, combined with the staples of applying and controlled viewing, with the occasional stop-and-think question to address some key points.

> Visually simulating a program by applying a given visualization engages programming beginners in a balanced way that is intended neither to be cognitively overwhelming nor to require an excessive time investment. VPS seeks to temper fragile knowledge rather than rely on it. It can be used in combination with other forms of engagement.

Last but not least, one practical weakness of presenting and creating visualizations is that humans are needed to give feedback on presentations and designs. In large introductory courses with limited resources, automatic assessment is very valuable (Section 10.1.4). Even if the use of visualizations is not graded or resources are available for grading by other means, the ability to have a system produce timely feedback to learners is a great boon. As the UUhistle system demonstrates, this is feasible on the applying level of engagement.

Which brings us to our next topic.

14.4 Tools make visual program simulation practical

It is possible to get a pen-and-paper approach to visualizing program states to work reasonably well. Gries and Gries, for instance, have used such an approach with some success (Section 10.3), although they,

too, later decided to develop a software tool (Memview from Section 11.3.2). Nonetheless, a system that facilitates VPS can be very valuable if not critical to the success of a VPS approach.

14.4.1 Tool support facilitates convenient practice

Some of the benefits of using software visualization systems are automation, consistency, and speed. A particularly noteworthy advantage of a program visualization system compared to pen-and-paper visualization and kinesthetic simulation in class is that a system makes it convenient and quick for learners to work on many example programs. Such practice can be done at any time, irrespective of class hours and at the student's own pace. The system provides a visualization so that the learner does not need to spend time drawing. Going back and forth in a dynamic visualization is simple in a tool but inconvenient on paper.

A programmer will only be able to program efficiently if he or she does not need to actively process the semantics of each language construct and how it solves a small subgoal of the complex problem at hand. The schemas corresponding to these low-level goals must be automated so that they do not tax working memory. Similarly, to form schemas on higher levels of abstraction, working memory must not be taxed by the details of each lower level. Research in cognitive psychology (Chapter 4) suggests that first forming and then automating a schema takes prolonged practice.

> Visual program simulation supported by a software system enables learners to conveniently practice execution-related topics. Deliberate practice helps learners form and automate the low-level schemas that complex activities build on.

14.4.2 A tool can give timely, automatic feedback

One of the benefits of a (bug-free) program animation is that unlike a human drawing a visualization, it does not make mistakes. All a program animation system does, however, is 'show how it really goes'. Program animation does not necessarily lead the learner to experience cognitive conflict (Section 6.5) between their conceptual structures (misconceptions) and the reality of the notional machine, unless the learner actively seeks to do so. A visual program simulation system such as UUhistle makes use of the computer's ability to automatically determine the correct answer without forgetting that the human tendency to err also plays a part in learning.

> Visual program simulation supported by a software system can be designed to allow learners to make mistakes in tracing a program. When mistakes occur, a system can help the learner (and the teacher) by giving automatic feedback in a timely fashion. Feedback tailored to the learner's own mistakes can help the learner to experience cognitive conflict between misconceptions and what is shown as the correct answer. An automated system can give feedback continuously, at each step of the simulation task.

14.4.3 Automatic assessment is a way to encourage engagement

Providing a learning resource to students is not the same as getting them to use it, and allowing students to engage with a visualization is not the same as getting them to engage with it. Asking CS1 students to use visualizations when tracing programs, or suggesting that the debugger is quite a useful tool, is not at all guaranteed to work. What is more, some students (and teachers, too) consider that stopping to cover underlying principles is a waste of time which could be better spent on writing programs, no matter how unlikely or fortuitous any success is without a viable understanding of those principles.

A crucial tool with which teachers can affect what students in formal education do – and what they learn – is assessment. If we want our students to learn the ability to trace the execution of programs in terms of a notional machine, then that is what they should practice and that is what they should be assessed on (cf. Biggs and Tang, 2007).

Just as a software system is capable of giving feedback about the execution of a program, it is capable of automating grading, and can be used to perform a part of the normative assessment of a programming course.

> Visual program simulation tasks can be normatively assessed by a software system. This makes possible the automatic assessment of knowledge about a notional machine. Assessment can play a role in getting students to pay attention to the visualization.

14.5 But visual program simulation has weaknesses, too

I have pointed up some positive aspects of VPS. Now let us look at a few negative ones.

14.5.1 VPS is not an authentic professional activity

Many forms of constructivism (Chapter 6) suggest that knowledge is highly context-dependent and the best way to facilitate learning is to have learners work on authentic tasks in complex, realistic contexts. Accordingly, learning to program should be a matter of working on complex, ill-structured, realistic, intrinsically meaningful programming projects, preferably in groups (Section 10.1).

The theory of situated learning (Section 6.6) is particularly big on these matters, and states that learning occurs through participation in a community of practice. Learning that is to be useful in a professional community should not be taught in a decontextualized school, but in that professional community, by making novices legitimate members of the community and having them participate in experts' work in some useful capacity, however peripheral.

Artificial simulation

Visual program simulation is not an activity that professional programmers engage in, and having novices perform it as part of their formal schooling is not a form of participation in the professional community. Many constructivists would undoubtedly worry about whether VPS will result in the development of skills that are only useful within the educational institution, perhaps only within the VPS system and its associated assessment mechanisms. Visual program simulation is precisely the kind of inauthentic practice that some would label "drill and kill" (but that does serve a purpose, as the previous sections have established).

Artificial assessment against well-defined correct answers also goes against the grain of those constructivists who question the idea of teachers setting standardized goals for their students in the first place. Visual program simulation is not in itself a way of enabling learners to create new programs whose quality and meaningfulness could be assessed by teachers and students in collaboration. Instead, it requires the learner to follow steps in a process that smacks of the sort of "knowledge replication" whose undesirability various constructivists underline (Section 10.1). From a constructivist point of view, VPS can nevertheless be seen as a form of scaffolding along the way to being able to create programs, which continues to be the main goal of programming education.

The challenge of contextualized content

Apart from the issue of real communities of practice, there is another matter that concerns the relationship between visual program simulation and programming context.

The easiest examples for a programming teacher to create are decontextualized toy programs that illustrate a point without being complicated by domain concepts and application code. As potential content for visual program simulation, such programs have the significant advantage that they have been pared down to the barest of essentials and involve a small number of simulation steps for the user to perform. This prevents the VPS exercise from becoming very long and tedious.

Using contextualized examples has clear advantages when seen from a constructivist point of view. Even if the programming course is not contextualized in a way that takes the students' specific future professions into account, using examples with real-world significance can be a great way to motivate students (see, e.g., Pattis, 1990; Media Computation teachers, n.d.). However, adding a realistic problem domain makes designing practical VPS exercises considerably more challenging. The example author will need to consider the trade-off between the benefits of a real-world problem domain and the 'clutter' that adding a domain brings into the visualization and the simulation task. Obviously, the issue is highly domain-dependent and also greatly affected by the level of abstraction used during VPS.

In mitigation, the decontextualized toy programs used in education do resemble the sort of 'foobar programs' that professionals actually write in authentic contexts, such as when learning a new programming language or trying out a new technology.

A trade-off between authenticity and direct engagement

The 2DET engagement taxonomy (Section 11.2) provides another framework for discussing task authenticity in visual program simulation, in contrast to other forms of program visualization.

When professional programmers use program visualization tools to visualize program dynamics, it is typically on the controlled viewing level of direct engagement, most commonly in a visual debugger. Different use cases involve different levels along the content ownership dimension of the 2DET, up to and including programs entirely written by the programmer.

Controlled viewing of various kinds of content is exactly the sort of engagement that most program animation systems for CS1 support (Section 11.3). Compared to these systems, visual program simulation involves a trade-off: it sacrifices the authenticity of controlled viewing in order to provide a higher degree of direct engagement and encourage learners to make use of the visualization.

A related limitation of visual program simulation – at least on a detailed level, as in UUhistle – is that VPS is effectively constrained to the lower levels of the content ownership dimension. Nothing actually prevents students from visually simulating programs that they wrote themselves, but this does not seem a very fruitful direction in practice. Not every program makes a good VPS exercise, and novices' solutions to programming problems often do not. Many learners would be quite annoyed by having to manually simulate a program after managing to write it, especially when using a program animator or debugger is also an available option should they wish to examine or present their creation's behavior.

Lessons learned?

In Sections 6.6 and 6.8, I presented critiques of the situated and constructivist theories of learning, some of which can be summarized as follows: complexity and authenticity are not always good; not everything needs or should be learned in context especially in high-tech fields; knowledge transfer is difficult but does happen; it makes sense sometimes to rely on a notion of correctness, especially when it comes to the technical reality of the computer; repeated practice is important for some learning. Even many critics agree that constructivist and situated theories can teach us important lessons about learning, but that extremist proponents of these theories take things too far.

What can we learn from the critical views of visual program simulation I have presented in this section?

An acknowledged weakness of visual program simulation is task authenticity. The risk of bringing about learning that fails to transfer outside the VPS system or outside programming courses must be taken seriously. Teachers must also be aware of the possibility of students questioning the legitimacy of VPS as a meaningful activity.[3] Authentic, motivating problems with real-world significance are important

[3] The empirical work that I will present in Part V supports the notion that knowledge formed during VPS can be transferred, but also underlines the fact that teachers need to support this transfer.

in programming education. As VPS does not address this need, teachers must cater for it in other ways, using a combination of complementary approaches.

Compared to the more common technology of program animation, VPS sacrifices some task authenticity for a greater degree of direct engagement between learner and visualization. Teachers must acknowledge this trade-off and also present learners with more authentic ways of using tools; again, a combination of different instructional strategies may be the best bet. One way to combine these modes of working with programs (in UUhistle, for instance) is to visually simulate teacher-given problems in order to learn about programming fundamentals and the visualization itself, and then make use of the same visualization in program animation mode to debug the learner's own creations.

I close the topic of task authenticity by noting that while the visual program simulation activity itself is not authentic, the activity that it is designed to teach about – program tracing – is very much so. Thinking about the execution of programs is something programmers really do; looking at visualizations without thinking about what they mean is not!

14.5.2 VPS involves a usability challenge and a learning curve

In visual program simulation, some of the complexity of program execution – which is usually taken care of by the computer so that we do not have to – becomes the learner's responsibility. Of course, the learner must not be required to take care of every single bit and byte, but needs to engage in the learning of key concepts. If the learner's task is oversimplified, VPS becomes mere program animation with an unusual interface. Nevertheless, VPS must be easy to use in order to be practical.

The interface of a VPS system must be easily learnable and convenient to use, and must not add too much extraneous cognitive load so as to detract from learning the content of the visualization. At the same time, it must be powerful enough to allow various manipulations of the notional machine. The task that the learner must perform is not very similar to anything that they are likely to have done in the past. Usability is a significant challenge.

Learning to use

Petre and Green (1993) found that inexperienced users of a visualization tend to get confused during a 'reading' of the visualization, and fail to make strategic use of it or even to notice what useful information is present. Ben-Ari (2001b), among others, has warned programming educators that non-trivial visualizations must be used in the long term so that users have time to become familiar with the tools used and the nature of the visualization. These concerns are highly pertinent in the case of a visual program simulation system such as UUhistle and its intricate controls.

A couple of isolated VPS exercises thrown into a CS1 course will have a significant overhead. It is also unlikely that a single short exercise will have much of an impact on its own. VPS probably needs to be used in a good number of exercises, preferably over a number of weeks at least, to be worth the bother.

Easing in features gradually as they become necessary (which UUhistle does; see Section 13.1.4 above) is a way of making the learning curve steeper.[4]

I discuss the usability of UUhistle in more detail in the next chapter.

14.5.3 VPS infantilizes the curriculum, Dijkstra would say

Visual program simulation will not help very much if you already have a dynamic perspective on program execution and know how to trace programs, and if programming concepts come to you easily. As we know from the early chapters in this thesis, this is not the case for many CS1 students. Nevertheless, we must not ignore those who come in with significant programming experience or who find it relatively easy to learn to program.

Ideally, only those students who need it and whom it helps would engage in visual program simulation. Fully succeeding in such an endeavor in any class let alone a large one would be quite a feat. Compromises

[4]Steeper in the positive sense of 'fast to make initial progress on'.

may be necessary; at the very least we should design VPS exercises in such a way that our stronger students are not wasting a great amount of time on something that merely bores them.

Consideration of the stronger students raises a number of questions. Am I making a mountain out of a molehill? Does VPS simplify what should not need so much simplification to be learnable, or worse, what must not be simplified? Does VPS in fact promote a naïve view of computing?

In 1988, a much-publicized and highly controversial manuscript by the celebrated computer scientist Edsger W. Dijkstra challenged the way computer science is taught. Dijkstra proposed a radical departure towards a CS1 that is very explicitly and formally grounded in mathematics and logic, and in which computers and notional machines play no part. While he was at it, Dijkstra singled out software visualization as a particularly ill-advised and contemptible form of curriculum infantilization that prevents students from embracing the radically different way of thinking that formal methods – and therefore also programming – require. Dijkstra's opinion piece and some of the ensuing critical debate were subsequently published by ACM (Dijkstra vs. al., 1989).

While the educational change Dijkstra advocated has not materialized, it is enlightening to stop to imagine what he would have thought about visual program simulation. I have attempted to do so in Figure 14.1. The interested reader may also wish to take a look at Tedre and Sutinen's (2008) overview of the three traditions of computing – mathematical, scientific, and engineering – and the implications of this tripartition for computing educators.

Figure 14.1: Facing Edsger W. Dijkstra in court

Bailiff: All rise. In the matter of the computing community vs. Sorva, the court is now in session, the Honorable Peter J. Denning presiding. Please be seated and come to order.

P. J. Denning: We are here to discuss charges of aggravated curriculum infantilization and computer use in the first degree. Prosecution, please proceed.

E. W. Dijkstra: We have been exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was such an obvious case of curriculum infantilization that its author should be cited for "contempt of the student body," but this was only a minor offense compared with what the visualizations were used for. They were used to display all sorts of features of computations evolving under control of the student's program! The system highlighted precisely what the student has to learn to ignore; we must expect from that system permanent mental damage for most students exposed to it.

Automatic computers are a radical novelty that one has to approach with a blank mind, consciously refusing to try to link history with what is already familiar. This the defendant, a true believer in gradual change and incremental improvements, is unable to see.

I have heard students and faculty speak of programs wanting things, knowing things, expecting things, believing things, etc. The anthropomorphic metaphor – for whose introduction we can blame John von Neumann – is an enormous handicap for every computing community that has adopted it. The analogy that underlies this personification is so shallow that it is paralyzing. Because persons exist and act in time, its adoption effectively prevents a departure from operational semantics and, thus, forces people to think about programs in terms of computational behaviors, based on an underlying computational model. This is bad because operational reasoning is a tremendous waste of mental effort. We must learn to work with program texts while temporarily ignoring that they admit the interpretation of executable code. We should reason about programs without even mentioning their possible "behaviors."

[Proceedings are interrupted by a deranged computer engineer who would rather continue to act as if it is all only a matter of higher bit rates and more flops per second. After a brief but spirited chase, he is caught and detained.]

E. W. Dijkstra: I would like to invite you to consider the following way of doing justice to computing's radical novelty in an introductory programming course. This is a serious proposal and utterly sensible.

It really helps to view a program as a formula. First, it puts the programmer's task in the proper perspective: he has to derive that formula. In order to train the novice programmer in the manipulation of uninterpreted formulae, we teach it more as boolean algebra, familiarizing the student with all algebraic properties of the logical connectives. The essential thing is that for whatever we introduce, the corresponding semantics are defined by the proof rules that go with it. Right from the beginning and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification. Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs.

[Proceedings are interrupted by a deranged mathematician who would rather continue to believe that Leibniz's Dream of providing symbolic calculation as an alternative to human reasoning is an unrealistic illusion. After a brief but bloody scuffle, she is caught and detained.]

E. W. Dijkstra: Confrontations with insipid "tools" of the "software visualization" variety have confirmed my initial suspicion that we are primarily dealing with yet another dimension of the snake-oil business. The defendant, like the "software engineering" profession as a whole, has accepted as his charter "How to program if you cannot."

If I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see a depressing picture. The universities will continue to lack the courage to teach hard science; they will continue to misguide the students, and each next stage of infantilization of the curriculum will be hailed as educational progress. I ask you, members of the jury, to convict this criminal, even if it is merely an exercise in futility whose only effect is to make him feel guilty.

The prosecution rests.

P. J. Denning: Thank you. The jury will now listen to the expert testimonies of several recognized computing professionals.

W. Scherlis: I am troubled by DA Dijkstra's suggestion that certain modes of thinking be avoided in problem solving. A successful problem solver will have a broad array of means at hand to tackle problems together with the maturity to make choices of which means are appropriate to circumstances. For the programmer, one of the means available is the use of informal operational intuition. Of course, this does not excuse failures to think intensionally when appropriate. It also does not excuse failure to come to grips with underlying computational models.

continues on next page

Figure 14.1 continued

M. H. van Emden: I think the course described by Mr. Dijkstra, where students derive programs from logic specifications into an unimplemented language, is a great idea. However, it will only work if the symbols manipulated have meaning for the players. And I do not see where that meaning can come from other than having messed around with programs that run or fail to.

J. Cohen: I feel that learning the foundations for writing sound programs ought to be fun and there should be a genuine sense of accomplishment when a student actually runs a program, finds unexpected errors, and corrects them. Our goal should be to provide not one but several approaches in teaching our students how to reason about programs.

R. Hamming: In the prosecution's statement, there is much "sound and fury", but only a fool would think it can be safely ignored. Apparently, reformers must often be extreme in what they say and do if they are to achieve a reformation.

One of the errors that Mr. Dijkstra's extremism has produced is that even if we tried to use the idea that programs should be "proved" by humans before they are run, the proofs are fallible. The idea applies to paper programs on a paper machine and not to reality. I also doubt that it is always wise to equate a program to a mathematical formula as he does.

Mr. Dijkstra flatly asserts that he knows "reality" and his opponents do not, but I put about as much faith in this as in the statement, "I am Napoleon." Indeed, in my opinion Mr. Dijkstra comes to grief simply because his "reality" is so far from most other people's.

T. Winograd: If DA Dijkstra is talking about the education of "computer scientists" in the narrow sense of theoreticians of formal computation, there is a bit of sense to his claim. However, I take Dijkstra's argument not to be just about the training of the small elite cadre of theoreticians, but, rather, about his hundreds of freshmen – the broad population of people who work with computing devices.

The DA must face the unpleasant truth that it is the job of someone to produce a collection of instructions that allow computing devices to function appropriately in practice. In some cases, this might be best done by producing a full formal specification and then converting that into code, but that methodology is debatable, at best, and far from universally applicable.

The DA's idealized view of programming shows its inadequacy when he says, "we should reason about programs without even mentioning their possible 'behaviors'." How can we do this and consider what is to happen in the drawing program I use if I drag the cursor outside of the window while in the process of specifying a rectangle? In other words, once we recognize that we are engaged in the design of operational computing devices, we must train people to think well in operational terms.

The primary subtext of DA Dijkstra's diatribe is a complaint about the lack of rigor in computer science education. I fully support his pleas that as educators we demand rigorous thinking, teach the beauty of mathematics, and encourage the virtue of facing uncomfortable truths. But, he confuses this with the claim that the essential part of computing education lies in the ability to manipulate formal abstractions, detached from considerations of operational devices, their behaviors, or their embedding in a world of people and activities. If he deludes his students into thinking this, they are in for a rude awakening when they try to function as computing professionals.

It would be foolish to ignore the value of the abstract mathematical skills the DA advocates, but it would be even more foolish to indulge the fantasy that they offer some magic that allows students to escape the hard work of learning about real computing.

P. J. Denning: Thank you, gentlemen. Does the defendant wish to make a statement?

Defendant: I do, Your Honor. Let me start with the charge of computer use. I confess to not promoting the use of formal proofs in the first programming course, and to endorsing the use of computers. However, these are not crimes. As the expert testimonies have pointed out, using computers in introductory programming is a good thing.

I do not accept the accusation of failing to nurture changes in thinking, although it is true that I do not endorse the particular change to introductory programming education that the DA is pushing for. Visual program simulation is one way of helping students significantly change their perspective on programs from a static one *to* a dynamic, operational one. Von Neumann and operational thinking may be at fault when it comes to some of the misconceptions that students have, but mathematics and declarative thinking also carry part of the blame. As for non-viable anthropomorphic metaphors of the computer, that is one of the very things that I seek to address through visual program simulation.

I plead not guilty to the charge of curriculum infantilization. As has been amply demonstrated, the need for better teaching of the runtime behavior of programs is real and has been identified as such by the computing education community.

P. J. Denning: Thank you. The jury will now retire to discuss the case.

I have not told the whole truth; this passage quotes and bastardizes Dijkstra vs. al. (1989).
This thesis has been in no way, shape, or form endorsed by any of the esteemed personages mentioned.

Chapter 15

UUhistle is the Product of Many Design Choices

The devil is in the details. Just because visual program simulation makes sense in theory does not mean it is easy to make it work in practice. The UUhistle system – our attempt to make VPS work – is a product of countless small design decisions. This chapter is a reflection on the way VPS is implemented in UUhistle.

Section 15.1 explains why we wished to create a program visualization system that is generic rather than specializing in certain concepts. In Section 15.2, I consider UUhistle's notional machine and the level of detail in the system, and their fitness for purpose. Section 15.3 explains how UUhistle is designed to address many different programming misconceptions. Finally, in Section 15.4, I examine UUhistle's usability in the light of established usability heuristics and rules of thumb.

Before reading this chapter, which goes into some detail about UUhistle, the reader should make sure they have either read Chapter 13 or experienced the system in action.

15.1 A generic visualization is a consistent platform for principled thought

We can group systems that visualize program execution into two categories. Generic systems (such as those reviewed in Section 11.3) visualize a generic notional machine that can deal with many language constructs. Specialized systems (see Section 10.3) visualize the dynamics of a particular construct such as pointers, parameter passing, or assignment.

Specialized systems have the advantage of being able to center on a topic and abstracting out all other aspects of the notional machine, for less clutter and a clear learning focus. They can also make use of visual tricks and metaphors that suit the particular topic especially well.

UUhistle is a generic system.

Advantages of generic systems

Generic systems have many advantages. An obvious one is that you can use a single generic tool to cover a lot of different concepts, rather than having to use a number of different systems. One implication is that the teacher who wishes to create their own example visualizations only needs to know one system in which they can create a variety of examples. Suitable specialized tools may also not be available for each topic of interest.

Even more importantly, learners do not have to learn to use many different visualization tools, many of which come with a learning curve of their own. Assuming that visual elements are used consistently within the generic system, learners have the further advantage of being able to anchor their understandings of new examples in their prior knowledge of the visualization and the notional machine it represents. Generic systems give a big picture of a notional machine which a motley combination of specialized tools will not provide. This intuitively pleasing idea is in agreement with many learning theories; fostering the

integration of new material with prior knowledge is stressed as important by constructivists, cognitive psychologists, and others.

Support for a generic system can be drawn, for instance, from conceptual change theory (Section 6.5), whether one subscribes to a knowledge-as-theory or a knowledge-as-elements perspective on conceptual change.[1] A visualization of a notional machine engenders conceptual change by occasionally cognitively conflicting with the learner's naïve knowledge. From a knowledge-as-theory perspective, the consistent theoretical framework of a generic visualization can be used to revolutionize the learner's naïve theory, replacing it with a viable model of a notional machine. From a knowledge-as-elements perspective, the generic visualization serves to provide a larger whole into which the student's existing knowledge can be integrated by highlighting underlying principles.

Research on the psychology of programming (Chapters 4 and 5) has shown that novices have many kinds of fragile, context-dependent, undergeneralized understandings of programming concepts, including program execution. They may have and unconsciously apply multiple and possibly contradictory mental models of the notional machine they are learning to control. Moreover, novice knowledge transfer tends to be based on superficial features rather than general principles, which is problematic. Consider parameter passing and assignment to non-parameter variables in Python, for instance. Although they look different in program code, parameters and assignment statements are both mechanisms for associating names with references. Both parameters and other variables are instances of the same concept (variable) and can be used in precisely the same way once initialized. This similarity is reflected in UUhistle's notional machine and its graphical representation of variables. Using one system for learning about assignment and another for parameter passing allows for different bespoke ways of visualizing these two concepts at the cost of not highlighting the similarities between them. Some novices struggle to make the connection and may end up with two context-dependent – and possibly contradictory and untransferable – mental models of these two parts of the notional machine.

Student-created content

Finally, an important practical advantage of generic systems is that they can be used with a variety of student-created content. Even though having students visually simulate their own programs is not likely to be a great idea (see Section 14.5 above), a hybrid simulation/animation system such as UUhistle is made all the more versatile by allowing students to explore self-selected aspects of self-selected (or self-written) example programs. This is something that only a generic system can accomplish.

15.2 UUhistle visualizes a useful notional machine

We designed UUhistle to visualize one particular notional machine. There were other options.

15.2.1 The notional machine supports two paradigms

UUhistle's notional machine consists of a particular set of elements (variables, stack, etc.) presented at a fixed level of abstraction, no matter whether the program shown is a purely imperative one or an object-oriented one (or a functional one, for that matter).

We chose this kind of notional machine for three reasons. The first reason is that the notional machine is useful not only for imperative but also for object-oriented programming. As I have argued in Section 10.4.3, object-oriented programming (in the sense in which the term is commonly understood) relies on two notional machines: a lower-level one that is an extension of an imperative machine, and a higher-level one expressed in terms of object interactions. It is the former kind of notional machine that UUhistle visualizes – necessary although not sufficient for genuine object-oriented programming.

The second reason for choosing this kind of notional machine was the good experiences and promising empirical results reported regarding the Jeliot 2000 and Jeliot 3 systems (Section 11.3), which visualize a similar notional machine for Java.

[1] My personal suspicion is that people have both naïve theories and context-dependent 'pieces' of naïve knowledge (with the latter being more common, possibly by a large margin).

The third reason was pragmatic: the CS1 for non-majors in which we were to try out our approach is a course in imperative Python programming that covers some basics of object-orientation near the end.

15.2.2 Concrete tracing is a natural choice

A mater-studiorum from Section 5.5. Programmers use two kinds of mental tracing strategies. Concrete tracing uses actual values, while symbolic tracing uses generic symbols (e.g., the value assigned to `x`, the return value of `foo()`). Symbolic tracing is lighter on working memory and is typical of the way people prefer to run mental models 'qualitatively'. Concrete tracing is needed in challenging situations, and novices need it often.

UUhistle's notional machine uses actual values to provide a form of concrete visualization, for several fairly obvious reasons. First, the computer uses concrete values when running Python programs. Second, novices need to learn concrete simulation to deal with difficult and common situations such as unfamiliar and buggy programs. Third, the external representation that is the visualization compensates for the failings of working memory.

Are there downsides to using concrete values? One might be that it leaves symbolic tracing untaught. However, it may be that people move quite effortlessly from the concrete to the qualitative/symbolic running of mental models when they have a solid enough understanding of the concepts involved, so perhaps symbolic tracing is not a skill that needs separate teaching.

15.2.3 There is not an unmanageable amount of detail

The developers of the Jeliot program visualization system specifically wished to show novices details about program execution that seem self-evident to experts (Ben-Bassat Levy et al., 2003). This serves an obvious purpose. However, when the level of abstraction is fixed, as in Jeliot, there is a very real risk that controlling the viewing of a detailed animation may turn into an endless 'click-a-thon' that merely bores students and distracts them from the meaning of the visualization. An occasional criticism of the Jeliot system is that the user has to view everything in detail without being able to focus on the aspect of execution that they are most interested in (Moreno and Joy, 2007).[2]

UUhistle is also all about exposing certain details of execution to novices and has a similar fixed level of abstraction to Jeliot, so the risk of excessive detail is there. Moreover, when UUhistle is used for visual program simulation rather than animating programs, the danger is even more ominous, as learners have to work to get past each execution step.

Excessive detail can be fought by learners themselves, by teachers, and by the system designer directly.

The learner vs. excessive detail

One way to fight excessive detail is by empowering the learner. Just like a debugger, UUhistle in program animation mode allows the user to have a program run until a specified line of code and set breakpoints (Chapter 13). The user can also choose to execute code one line at a time, reducing the number of interactions needed to get to an interesting point in the program run.

The teacher vs. excessive detail

A visual program simulation exercise does not allow the user to skip freely to an arbitrary stage of execution. Wherever the user has to manually simulate the program's execution in UUhistle, he does so at the lowest level of granularity. However, with careful VPS exercise design, the teacher can allow the user to skip past boilerplate code and other unnecessary detail (Chapter 13). The teacher can configure exercises so that some code is executed automatically before the VPS task starts (either by making it 'hidden' or by defining a preset breakpoint); there are also many ways of fine-tuning simulation/animation hybrids.

[2]The newest version of Jeliot supports a "Run until line" command that somewhat alleviates this problem if students find and use it.

The system designer vs. excessive detail

The system designer can fight excessive detail is by automating or altogether eliminating any execution steps that are of secondary importance to the learning goals of the system. UUhistle shows a lot of detail, but not all; although a VPS exercise requires the user to do quite a lot manually, it certainly does not require them to do everything. Chapter 13 already gave an idea of the animation steps UUhistle shows and the simulation steps the user needs to perform in a VPS exercise. Now let us consider what the user does *not* see in UUhistle and what they do *not* need to do.

Mayer (e.g., 1979, 1985) uses the term *transaction* for the kind of operation that corresponds to a primitive operation of a notional machine right above the machine (or hardware) level:

> *Transactions are not tied to the actual hardware of the computer; however, they are related to the general functions of the computer and are the building blocks from which statements are made. A transaction is a unit of programming knowledge in which a general operation is applied to an object at a general location.* (Mayer, 1979, p. 590)

In the notional machine Mayer uses for teaching BASIC, executing the statement `LET B = A + 1` requires no less than ten transactions (Mayer, 1985):

1. find the number literal in the program code;
2. store that number in a temporary memory area;
3. find the number stored in `A`;
4. store that number in the temporary memory area;
5. add the two numbers;
6. find the location of variable `B`;
7. erase its value from memory;
8. put the sum into variable `B`'s location in memory;
9. go on to the next statement;
10. do what it says.

Representing all of these operations as separate execution steps in UUhistle would make stepping through animations – let alone visually simulating programs – unbearably tedious. Compared to Mayer's transactions, UUhistle simplifies things in several ways. Reading code and locating objects and variables in memory are done automatically by the computer when animating and visually by the user (when VPSing); they are never separate execution steps. Variable values get replaced by new ones automatically as part of the assignment step. On default settings, moving to the next line is shown as a separate step when animating, but happens automatically when simulating (except when branching). The decision to keep going with the fetch-execute cycle as each new statement is reached is implicit.

Beyond the above single-statement example, too, there are many things that UUhistle does for the user automatically, including creating 'simple values' in the heap (integers, strings, etc.), storing return locations in stack frames, and deallocating frames. All these happen without any input from the user or a separate execution step. Following a reference to access an object's attributes is also automatic when animating, and the learner uses his or her eyes for dereferencing when simulating.

Although some students may be initially puzzled by the way 'simple values' appear in the heap automatically – and it is good to explain this behavior to them – the creation of these objects is so common in all Python programs that automation is necessary to keep the visualization practical.

The selection of the simulation steps that the learner needs to perform in a VPS exercise reflects our intention to allow the learner to focus more on progressing forward and worry less about memory cleanup. The learner's progress is kept track of by the 'program counter' (that is, the highlight on the current line). Requiring the learner to change lines manually seems excessive for most purposes, as we expect that even novices generally correctly know that they are done with a line of code when they really are, at which point it is reasonable to have the line change happen automatically.[3]

Finally, the way UUhistle allows the learner to take 'shortcuts' past certain routine simulation steps (see Section 13.4) is a practical compromise between consistency and convenience.

[3] Probably a more common problem is that the learner thinks they are done with a line when they are not.

15.3 UUhistle is designed to address known misconceptions

Visual program simulation in UUhistle is based on the idea that the learner needs to locate the next execution step. To encourage the learner to think, there need to be many GUI operations to choose from even if only one of them is the correct one. Allowing many 'wrong paths' to be taken reduces the effectiveness and, presumably, the appeal, of mindless trial and error.

Allowing incorrect operations also provides a way of addressing students' misconceptions about program execution and language constructs.[4]

15.3.1 A VPS system can be misconception-aware

Ben-Ari (2001b) leavens our enthusiasm with a cognitive constructivist point of view:

> *For the use of* [an educational software visualization system] *to be effective, the teacher must (a) have a clear idea of the existing mental model of the student, (b) specify in advance the characteristics of the mental models that the instruction is intended to produce, and (c) explain exactly how the visualization will be instrumental in effecting the transition. This is an immensely difficult undertaking.* [For one thing,] *the existing mental models of the individual students are different and it takes quite a lot of effort to elicit even an approximation of a cognitive structure.*

Ben-Ari's statement underlines the usefulness of knowing about the mental models and misconceptions of individual students. Eliciting these from each student is indeed immensely difficult. A couple of things make life easier for the visualization designer, however. First, nature constrains our knowledge-constructing activities (see Section 6.3 above), so it is reasonable to expect there to be similarities between learners' mental models and certain misconceptions to be more common than others. Second, creating a state of cognitive conflict does not necessarily and always require a clear idea of the learner's mental model. In some cases, it is surely possible to make the learner deal with a conceptual model (visualization) so that the learner him- or herself is in a position to discover a discrepancy between the conceptual model and his or her own prior mental model, whatever it is like.

That having been said, Ben-Ari's point is an important one. A visualization that directly and explicitly addresses a learner's misconception probably has a better chance of success than one that does not.

How can a fully automatic visualization system possibly address individual students' misconceptions? A visual program simulation system may fare better than most.

VPS and misconceptions

Here is one way of looking at how a VPS system may address a specific misconception that concerns program behavior.

Level 0 **By doing nothing.** The system does not address the misconception. Nothing in the visualization conflicts with the misconception.

Level 1 **By showing what's right.** The system shows that what actually happens when the program is run does not match the learner's misconception. If the learner pays enough attention, they may experience cognitive conflict between their understanding and the visualization.

Level 2 **By having the learner do what's right.** The VPS system requires the learner to simulate aspects of program execution that pertain to the misconception. The system *prevents* making the kind of mistake the learner would probably make if they consistently followed through with their misconception. The learner has to perform simulation steps that go against their misconception. The system does not encourage the learner to contrast the accepted (correct) simulation step with any alternatives.

[4] Substantial parts of Section 15.3 are replicated, with the publisher's permission, from an earlier publication: Juha Sorva and Teemu Sirkiä: "Context-Sensitive Guidance in the UUhistle Program Visualization System", In *Proceedings of PVW 2011, Sixth Program Visualization Workshop*, pages 77–85, Darmstadt, Germany, Technische Universität Darmstadt, 2011.

Level 3 **By allowing what's wrong.** The VPS system requires the learner to simulate aspects of program execution that pertain to the misconception. Further, the learner *can* follow through with their misconception by taking an incorrect simulation step that corresponds to it. To solve the VPS exercise, the learner has to pick the correct execution step rather than one that matches their misconception. If the learner fails to do so, the system notifies the learner that something is wrong. Ideally, this alerts the learner to the shortcoming in their thinking, even though the system cannot guess what the specific problem is.

Level 4 **By discussing what's wrong.** As above, except that the system gives the learner feedback that is tailored to the kind of misconception that the learner's mistake probably indicates. The system helps the learner reflect on the particular misconception.

By presenting this scheme, I have sought to enumerate different ways of addressing misconceptions, not to suggest that every misconception needs to be addressed at Level 4. Showing carefully selected program examples will sometimes suffice to address a misconception. Misconceptions can also be addressed during program-writing tasks, for instance. Not every misconception even *can* be addressed at the higher levels – the learner's understanding may be so vague or inconsistent that they cannot follow through with it in a VPS exercise.

Nonetheless, it seems a reasonable conjecture that for many misconceptions, each higher level in this scheme is more likely to result in fruitful cognitive conflict than the lower ones.[5]

A program animation that does not activate the learner and has no clue about what the learner thinks can only get to Level 1 in the hierarchy. Visual program simulation can address misconceptions on all these levels, as the following examples show.

15.3.2 UUhistle addresses different misconceptions in different ways

Appendix A is a catalogue of over 150 introductory-level programming misconceptions reported in the literature. I reviewed the studies that reported these misconceptions in Section 3.4. Here, I use the word misconception in a very broad sense to refer to many kinds of understandings that are likely to be non-viable in many CS1 contexts. This includes not only understandings that contradict commonly accepted definitions and the actual behavior of computers in more or less specific ways, but also partial understandings and vaguely reported 'difficulties' with a particular concept.

We designed UUhistle to address many of these known novice misconceptions at high levels of the above hierarchy. Some others it addresses only at lower levels. I will give examples below.

Level 0: doing nothing

UUhistle addresses a few of the misconceptions listed in Appendix A at Level 0, that is, not at all. An example is the misconception that the computer keeps program output in memory as part of program state. This particular misconception is, in fact, one that UUhistle and other program visualization systems may encourage, as they show a visualization of program state alongside output.

Also at Level 0 are the misconceptions in which the learner thinks it is possible for the programmer to do something that is not actually supported by the language used, e.g., to write a method that replaces the object itself with another. UUhistle cannot demonstrate that such instructions do not exist. Sometimes an increased understanding of what existing instructions do, which may be gained through VPS, may still indirectly show that the nonexistent instructions are unnecessary for the learner's intended purpose.

Level 1: showing what's right

UUhistle's VPS exercises address some of the listed misconceptions at Level 1. Most of these have to do with control flow, which is largely automated in UUhistle on default settings.

[5] It is not my intention in this thesis to test this assumption. My primary aim here is just to illustrate different ways in which a VPS system can address misconceptions, not to rank them.

Consider the misconception according to which whenever the conditional part of an `if` statement is executed, anything that comes after the statement is not. This is shown to be incorrect by UUhistle's visualization, which the learner may observe from the way control moves from line to line. After the learner is done with executing the conditional part, UUhistle chooses the next line automatically. The learner is not in any way involved in the act of changing lines. This misconception, like many control-flow related misconceptions, can be addressed at a higher level in a VPS exercise that is configured to require the user to change lines manually.

Level 2: having the learner do what's right

UUhistle can address roughly half of the misconceptions in Appendix A at Level 2, by forcing the learner to simulate the correct behavior of programs.

Consider the misconception that a variable can hold multiple values at the same time. This is in direct conflict with what happens in UUhistle when the user drags a new value to a variable. No matter how much the learner might expect to produce a variable with two values, it will not happen as a result of the drag-and-drop operation, nor will they find another GUI operation that makes it happen.

Another, more generic misconception is some learners' failure to see that a program has a dynamic existence in addition to its static aspect. UUhistle's VPS exercises address this by requiring the user to simulate program dynamics, without allowing any alternative way of dealing with the program merely as code.

A class of misconceptions incorrectly constrains the capabilities of the programmer. For instance, the learner may believe that the attributes of simple objects must always be accessed through a composite object. Arguably, the main way – and often a sufficient way – to address such misconceptions in teaching is to select examples so that they show variation in the ways in which the task can be accomplished. What VPS can add is the requirement for the learner to think their way through the execution of the counterexamples. A similar sort of misconception is the excessively narrow definition of a concept, e.g., an object may be seen as just a wrapper for a single variable (Holland et al., 1997); here, too, VPS can require the learner to work through an example that shows that objects can be more than that.

Level 3: allowing what's wrong

Several dozen of the misconceptions in Appendix A are addressed at Level 3 by UUhistle. Here, the system allows a misstep that matches the misconception, but signals that there is a problem. Automatic feedback points out that a mistake was made, and may give some generic advice, but does not directly address the specific misconception.

For instance, the simple misconception that the assignment of simple values works in the opposite direction is reflected in the learner dragging a value in the wrong direction. UUhistle allows the learner to carry out this incorrect step and informs them that they need to think again. UUhistle v0.6 does not, however, address the misconception that (possibly) led to the mistake by specifically explaining the right-to-left unidirectionality of assignment statements.

Also at Level 3 is UUhistle's response to the misconception that values are updated by the computer according to their logical context. The learner can try to reflect this understanding in how they simulate a program – by updating variables at the wrong times – but will get an error message. Similarly, there are some misconceptions in which the learner thinks that the programmer does not need to do something (e.g., call constructors, store return values), because that something is somehow automatically and implicitly taken care of without explicit instructions to do so. A suitably designed VPS exercise in UUhistle allows the learner to carry out these implicit steps at whichever point of execution the computer supposedly does them, and will inform the learner that they made a mistake.

Level 4: discussing what's wrong

UUhistle v0.6 addresses a handful of misconceptions at Level 4. UUhistle attempts to identify these misconceptions and gives tailored feedback when the learner appears to be under one of them.

For instance, the learner may fail to create another frame for a recursive call and instead reuse the current frame's variables. This mistake may reflect a "looping model" of recursion. UUhistle remarks

about this mistake to the user in a particular way, explaining the need to create new frames for each recursive call. Another example is the misconception that assigning an object (reference) creates a copy of the assigned object. Figure 13.11 on page 206 shows how UUhistle handles this second example.

Reaching the highest levels

Adding more Level 4 guidance for the user is one of the main priorities in UUhistle's development for the near future. In our tentative estimate, at least several dozen of the misconceptions catalogued in Appendix A look promising in the sense that we can make a reasonable guess as to what the user is likely to do in VPS if they consistently follow the misconception. These misconceptions could perhaps be detected fairly reliably by UUhistle from the learner's VPS actions, and could be addressed automatically with context-sensitive, misconception-aware feedback. An example is the assigning-in-the-wrong-direction misconception mentioned above, which is presently addressed at Level 3.

15.4 UUhistle is fairly user-friendly

The following is an informal commentary on the usability of UUhistle's VPS exercises and some trade-offs in the system's design.

We – UUhistle's creators – are not usability experts. The purpose of this section is to expose our thinking for criticism and to highlight what we think are relevant issues to other creators of program visualization systems who wish to do either something similar or something better.

This section focuses on VPS exercises as seen by students. I will not discuss, for instance, the animation controls, the pitfalls of having a separate mode for interactive coding, or the many improvements that might be made to UUhistle's teacher interface.

15.4.1 The user interface is not maximally simple. . . and should not be

We have tried to design UUhistle to be easy to use. We have not tried to design it to be extremely simple.

Devices that are meant to be easy to use, and products that are meant to appeal to their potential users, do not *necessarily* need to be simple (see, e.g., Norman, 2007, 2008, and links therein). Simplicity is one way towards usability, but not the only one.

> *The world is complex, and so too must be the activities that we perform. But that doesn't mean that we must live in continual frustration. No. The whole point of human-centered design is to tame complexity, to turn what would appear to be a complicated tool into one that fits the task – a tool that is understandable, usable, enjoyable.* [. . .] *most important of all is to provide an understandable, cohesive conceptual model so the person understands what is to be done, what is happening, and what is to be expected.* (Norman, 2008, pp. 45, 46)

In a learning tool especially, maximal simplicity of user interactions is not a goal in itself. We want convenience, yes, but convenience of successful learning. We want users – learners – to take a bit of time to engage with the system and to have a bit of trouble of the right kind while avoiding unnecessary complications and excessive cognitive load.

The user interface of UUhistle's VPS exercises is somewhat complex, but we have aimed for that complexity to be understandable. Executing a program on UUhistle's general-purpose notional machine is an intrinsically complicated task that is usually automated (simplified!) but whose complexity is brought to the fore by the VPS exercise. We want learners to stop and think about the visualization and the simulation task. The design challenge is to accomplish this while minimizing the extraneous difficulties created by the user interface.

We are encouraged by recent work on spatial algorithm visualization which suggests that programming students can cope with simulation exercises that involve fairly complex visualizations and GUI operation semantics (Nikander et al., 2009).

15.4.2 UUhistle's VPS exercises strike a balance between cognitive dimensions

Green and his colleagues have developed a set of *cognitive dimensions of notations* (see, e.g., Green and Petre, 1996; Green and Blackwell, 1998; Green, 2000; Green et al., 2006). These dimensions serve as a vocabulary for discussing cognitively relevant aspects of artifacts, and are intended as a convenient and inexpensive "broad-brush evaluation technique" during the design of user interfaces and notations. An influential paper by Green and Petre (1996) applied the dimensions to visual programming; the authors and others have applied them to many other domains (see Green et al., 2006).

Table 15.1 names and outlines 14 of Green's cognitive dimensions, and gives my interpretation of what the dimensions mean in a VPS context. Slightly different versions of the cognitive dimensions framework have been published in different articles; Table 15.1 is based on Green and Petre (1996), Green and Blackwell (1998), and Green (2000).

I will now discuss how UUhistle's visual program simulation exercises are located in these cognitive dimensions. Out of necessity, my commentary (like UUhistle's design) is partially based on conjecture as an empirically grounded psychology of visual program simulation does not yet exist.

When reading what follows, the reader will observe that – as Green and his colleagues have underlined – the dimensions are interdependent and involve many complex trade-offs. For instance, allowing the user to define abstractions can have the desirable effect of reducing viscosity but may also introduce hidden dependencies. The relative importance of each dimension depends on the task.

Closeness of mapping

Even someone who knows very well how Python programs work will need to learn a few tricks in order to simulate programs in UUhistle. Figuring out how the evaluation areas work and finding where to click to create new elements of different kinds are perhaps the trickiest of the tricks.

However, most aspects of a VPS exercise map directly to deeper meanings. There is very little in UUhistle's display that does not closely correspond to a concept in the notional machine that UUhistle teaches about. The simulation steps that the user performs have been chosen to correspond directly to what the notional machine does as it executes a program.

Not all of the 'games' the user has to learn to play are notation-induced, and neither are they necessarily bad. For instance, that a function call requires the creation of a frame and some local variables is part of what users need to learn and what they may struggle with. However, such a struggle is not an undesirable side effect of UUhistle's notation – unless it is the case that the notation makes it hard to realize the need for frames and local variables – but part of an intended learning activity.[6]

Novice programmers do not yet have a clear idea of the domain that UUhistle's representation maps to (the notional machine). When they use UUhistle, they learn about representation and domain at the same time. Consequently, they may not realize whether they struggle with a superficial GUI issue or are failing to understand a deeper concept. I will return to this important consideration in Part V.

One of the main reasons why not too many 'simulation games' need to be learned in UUhistle is consistency.

Consistency

Internal *consistency* has been the driver behind many of the design decisions made during UUhistle's development. As I explained in Section 13.4, there are few different kinds of GUI operations that the user needs to learn, and each kind of operation is consistently used to accomplish a particular kind of simulation step. For instance, once the user has learned that arithmetical operators are first dragged to the evaluation area and then clicked on to execute, it is not difficult to transition to calling library functions by first forming the function call in the evaluation area and then clicking on it.

There are limitations to UUhistle's consistency, some due to the notional machine, others due to choices we made regarding representation and convenience of use. One limitation is that most visual

[6] One of the additional cognitive dimensions proposed in the literature to extend Green's original set is "useful awkwardness", which forces the user to reflect on their task, seeking to produce an overall gain in efficiency (Blackwell, 2000). One way to look at visual program simulation is that it promotes useful awkwardness for the purpose of learning.

Table 15.1: Cognitive dimensions of visual program simulation

Dimension	Description	Key questions in VPS
Closeness of mapping	closeness of representation to problem domain	Are there any graphical elements that are there only for the sake of the simulation activity? What 'simulation games' do you need to learn that do not teach about programming concepts?
Consistency	similar semantics expressed similarly	When something about the simulation GUI is known, is it possible to guess the rest?
Visibility	the ability to view components easily	Is it easy to get any part of the visualization into view?
Diffuseness	number of symbols or amount of space used to express a meaning	Do graphical elements take more space than necessary?
Role-expressiveness	the purpose of a component (with respect to an overall plan or a different notational layer) can be readily inferred	Is it easy to tell what aspect of the given program's execution a visual element represents?
Secondary notation	facilities for the user to express information using secondary means (e.g., color, formatting, annotations)	Can the user move, group, or annotate visual elements in ways that they find useful but which are not part of the simulation?
Hidden dependencies	important links between entities are not visible; changing one may have unexpected repercussions on another	Can performing a simulation step impact on a part of the visualization that the user is not presently focusing on?
Hard mental operations	operations that – because of the notation used – become difficult to work out in one's head when used in combinations	Are there execution steps or visual elements that the VPS system represents in a confusing way when they appear in combinations?
Progressive evaluation	work-to-date can be checked at any time	Can the user already get feedback on their actions during the simulation?
Error-proneness	the notation invites careless mistakes and the system gives little protection	How likely is it for the user to make a careless mistake even when they understand the program code, and for that mistake to go unnoticed?
Premature commitment	the user has to make choices before all the necessary information is available	Does the user need to revisit previous decisions as more information becomes available?
Provisionality	support for sketching, exploring alternatives, and playing 'what if' games	Can the user explore what would result if the notional machine behaved differently?
Abstraction	the role of user-defined abstraction mechanisms in the system	Does the system preclude, allow, or require user-defined ways of grouping related simulation steps?
Viscosity	resistance to change; effort needed to accomplish a goal	Does it take many repetitive simulation steps to accomplish a goal that is thought of as a single action?

components (variables, values, classes, functions, etc.) look similar – like boxes with rounded corners – but some of them are used differently from others. Variables cannot be dragged and dropped, for instance, and only some element types (function and method calls, operators) can be clicked to execute them, and then only while they are in an expression evaluation area. Elements lock in place after being dragged and cannot be dragged back to undo operations.

We have used various small tricks to partially compensate. For instance, changes in mouse cursor signal what can be dragged, possible drop locations are highlighted during a drag operation, and tooltips suggest what can be clicked. Colors, sizes, and locations are the main ways of distinguishing between different kinds of 'rounded boxes'; tooltips help, too. For example, all variables in UUhistle have the same color (green). All values have the same color (light blue). The uniform use of color emphasizes how all variables in Python are similar, no matter where they are located or what type their contents have, and how all values of different types are dealt with in the same way.[7]

Green and Petre (1996) caution us that consistency depends on the user's understanding of the structures involved. What seems consistent to the designer given his understanding of a notional machine and the structure of a GUI may well not match the user's mental models. Given that we hope UUhistle to be usable by untrained users who are novice programmers, the reader should take my claim of consistency with a pinch of salt.

Visibility

A visual program simulation system with high *visibility* allows the learner to view and access any part of the visualization at will. UUhistle is built for very high visibility: the small programs typical of VPS exercises and their small data sets often fit nicely on a typical display so that everything is readily visible. When space runs out, the user can resize areas and use scrollbars that appear automatically within each panel as needed. The user never has to open up new views or expand components to gain access to something.

Complex programs use more memory, require more graphics, and limit visibility somewhat. However, since VPS exercises are meant to be short and to the point anyway, visibility will not usually be a serious problem in practice.

Diffuseness

As noted above, nearly all of UUhistle's user interface is devoted to representing the state of a notional machine. Is this accomplished with a minimal set of graphical elements and a minimal use of space? Green and Petre (1996) concede that assessing *diffuseness* is challenging.

There does not appear to be a significant problem with diffuseness in UUhistle. As noted, there is enough space to visualize simple programs in VPS exercises. The number of graphical elements is comparable to that in the Jeliot 3 system (Section 11.3.2), which has proven to be usable by novices. One difference is that UUhistle shows all the frames in the call stack simultaneously, whereas Jeliot and typical debuggers show only a single frame at a time. The use of more space and more visual components pays off in heightened visibility and role-expressiveness.

Role-expressiveness

Green and Blackwell (1998, p. 41) explain *role-expressiveness* as follows:

> *This dimension describes the ease with which the notation can be broken into its component parts, when being read, and the ease of picking out the relationships between those parts. An experienced electrical engineer can look at a radio circuit diagram and quickly pick out the*

[7] Ford (1999) studied how second-semester programming students visualized C++ programs, and reports that students are fond of using different colors or shapes to distinguish between different types of variables in situations where type information is important. Python variables are not typed, however, and having a variable 'change' as it is assigned a differently typed value could give students the wrong idea. We did consider having different shapes for differently typed values, but decided (for now) against it, as we were not convinced that the benefit would outweigh the additional complexity of the visualization. We do not expect that not being able to tell the type of each value is a major challenge to using UUhistle.

parts that deal with different stages of the process (detecting the signal, first amplification stage, etc.).

In my interpretation, role-expressiveness in VPS primarily concerns the clarity of the mapping between the program code and the visualization of its execution. Ideally, it should be easy for the user to tell what the purpose of any graphical element is in the current program run, that is, how it serves the overall purpose of making the given code do what it does. For instance, given a program and a visual snapshot of its execution, it should be easy to tell that certain elements are used to store the state of certain function calls currently in progress.

UUhistle addresses this issue by littering the visualization with more and less obvious clues, which include the following.

- The current line of code is always highlighted, showing that the current simulation steps are concerned with the execution of that line.
- Identifiers used in the code also appear in the visualization, drawing explicit mappings between entities in memory and the program code.
- The expressions in the evaluation area use symbols (operators, parentheses, etc.) that match Python syntax.
- The different panels in the UUhistle GUI are named to indicate their different roles in the notional machine.
- Colors are used to distinguish different kinds of elements (function definitions, objects, variables) from each other. Some objects (function objects and class objects) are grouped in their own areas in the GUI to emphasize their particular natures.
- The call stack is shown in its entirety. In each frame, the currently active function call is highlighted to clarify which part of a complex expression is currently being executed (see, e.g., Figure 13.1 on p. 194). This underlines the relationships between the frames in the call stack and makes explicit the stages of expression evaluation.

Moreover, visual program simulation as an activity demands that the user constantly pays attention to both code and the visualization of program state so as to know what to do next. Because of this, we may conjecture that the user is likelier to be aware of the relationships between code and visual elements than when merely looking at a visualization.

Role-expressiveness, too, is difficult to measure. The above can perhaps be taken as an indication that UUhistle is fairly role-expressive.

There are weaknesses, too. One is that since most visual components look similar, one has to rely on the texts, colors, and layout to figure out what is what. Contrast this, for instance, with the very different graphics of Sajaniemi's metaphorical program animations (Figure 11.17 on p. 166).

Another weakness is that UUhistle only highlights the current line of code in its entirety and does not show which part of the line is being executed. This is because it is the learner's job during VPS to determine the order in which the execution steps at each line are to be carried out. The downside is that sometimes it may be unclear (because forgotten by the user) how a value came to be in the evaluation area or which parts of a line's execution have already been dealt with. There is an elevated risk of this happening when control returns from a function or method call to a partially executed line. This problem is likely to be a minor one, however, and UUhistle's support for undoing and redoing (to bring to mind what has just been going on) alleviates it.

A final remark on role-expressiveness. The user cannot understand the role a visual element serves in expressing program execution unless they know that such a role exists. The "What is this?" links are designed to help novices learn about what the visual elements mean, as is the Info box (see Section 13.1 above).

Secondary notation

UUhistle's VPS exercises simply do not support secondary notation. The user cannot color, move, group, label, or otherwise annotate any of the visual elements in any way beyond the VPS task proper. The use

of color, the layout of objects and variables, and all other aspects of the visualization are strictly under the control of the system; UUhistle uses them in an attempt to bring about a particular interpretation of the visualization and enhance role-expressiveness (see above).

If the purpose of UUhistle were different, the complete lack of support for secondary notation could be a problem. However, since it is intended to be used by novices with a shaky (albeit improving) understanding of the content of the visualization, this lack might even be a blessing. Someone with a poor understanding of the concepts involved is not likely to make efficient use of secondary notation and might use it to produce representations that are misleading; Green et al. (2006) conjecture that novices may benefit from a system that minimizes secondary notation. There is also a real danger that learners would confuse any facilities for using secondary notation with the actual simulation steps. Given the further consideration that secondary notation tends to increase viscosity – already a somewhat problematic dimension for UUhistle (see below) – I conclude that UUhistle is probably better in this respect as it is.

Hidden dependencies

In a VPS exercise in UUhistle, not much happens without the direct involvement of the user, and most of what happens happens right where the user's attention is expected to be: at the mouse cursor and in the program code panel. Simulation exercises themselves do not, then, suffer from major hidden dependencies. There is one exception: built-in functions may have non-obvious side effects when applied. For instance, executing the `print` function causes output to appear in the console below. We are planning to highlight side effects visually in future versions of UUhistle.

Some hidden dependencies are intrinsic to Python and programming.[8] UUhistle cannot eliminate these dependencies (and must not, since the user is expected to learn to live with them), but can assist the learner with them. For instance, changing an object's state through a reference impacts on every other part of the program in which that object is used through any reference. As shown in Figure 13.4, UUhistle tries to draw learners' attention to this kind of hidden dependency, which is known to be problematic for many novices.

Hard mental operations

"Hard mental operations" are operations that are not intrinsically difficult aspects of what is being expressed, but are instead made difficult by the notation used. Further, they have the property that even though a single occurrence of the operation is easy to understand, a combination of even two or three such operations greatly complicates things. When faced with such combinations, users often experience cognitive overload and resort to strategies such as making notes on paper and tracing visualizations with their fingers (Green and Petre, 1996).

I am not aware of UUhistle's visualization introducing any hard mental operations in this sense.

Progressive evaluation

In a VPS exercise, UUhistle not only progressively assesses the user's actions at each step but provides automatic and instantaneous feedback on any mistakes the user makes. The user does not have to wait until the end of the program is reached to find out if their solution works.

Giving feedback only at the end of a simulation exercise would not be a user-friendly option and would punish the learner excessively for mistakes. Constant progressive evaluation has obvious benefits. However, from a learning point of view, there is also a potential drawback: instant feedback given at each simulation step may encourage mindless trial-and-error strategies. We have considered various midway alternatives between these two extremes, but no alternative feedback-giving strategies are currently implemented in UUhistle.

The availability of constant automatic feedback also significantly impacts on several of the remaining cognitive dimensions.

[8] Python as a program-authoring language is an entirely different artifact to which the cognitive dimensions can be applied.

Error-proneness

A VPS exercise in UUhistle involves many details. There are opportunities for careless mistakes even if one 'knows what one is doing'. Users may absent-mindedly perform simulation steps in the wrong order. Values may be dragged accidentally to the wrong places. Steps such as defining parameter variables may be passed by in haste.

All this has the potential to be very troublesome. However, UUhistle is redeemed by its instant automatic feedback: mistakes are not left unnoticed.

Premature commitment

Instant feedback also means that there is no need for the user to correct earlier decisions that have already been assessed as correct. Neither does the user, in the role of a deterministic machine, need to make any premature guesses. The necessary information for choosing each simulation step is available in a timely manner.

Provisionality

UUhistle supports the provisional 'sketching' of answers to VPS exercises, yet does not.

In principle, it is possible for the user to 'see what would happen if it was done this way'. The user can use UUhistle to explore alternative paths that execution might take if the notional machine worked differently. For instance, the student could use a looping model of recursion instead of creating new frames on the stack for each recursive call (running into an eventual problem, assuming a non-tail-recursive program). Easy-to-access undo and redo commands also support provisionality.

In practice, however, UUhistle does not encourage such use. On the contrary, the instant negative feedback for deviations actively discourages it.

Encouraging provisionality would help learners follow through with their misconceptions and come to realize their non-viability. It could also allow for other 'mind games' that would contrast the notional machine being taught with other notional machines. In our case, however, the practical usefulness of instant feedback outweighs these intriguing possibilities.

Abstraction

Green and Petre (1996) define an abstraction as "a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure". They group systems into three categories on the basis of their relationships to user-defined abstractions.

An *abstraction-hating* system does not allow the user to define macros, templates, functions, concepts, or other abstractions.

An *abstraction-hungry* system requires the user to define abstractions in order for the system to be useful. The price paid for greater expressive power is a more challenging learning curve. Novice users especially may struggle to form useful abstractions and may be put off by a system that requires them to be defined. Abstractions may also introduce problems with hidden dependencies.

An *abstraction-tolerant* system supports but does not demand user-defined abstractions.

In these terms, UUhistle is an abstraction-hating system. The user cannot, for instance, form macros that correspond to the execution of several execution steps at once. The learning curve is one reason for this. Spending a significant amount of time learning to build macros within a VPS tool seems less than optimal. Novice-designed abstractions might also accidentally 'make things too easy' and lead the user to ignore important details that UUhistle's pedagogy seeks to emphasize. On the downside, UUhistle's lack of support for user-defined abstractions contributes to its viscosity (see below) and limits the effectiveness of experienced users.

Despite the drawbacks, the abstraction-hating nature of UUhistle seems reasonably justified.

Viscosity

It is useful to distinguish between two different kinds of viscosity in VPS exercises:

1. the amount of work needed to initially accomplish a goal using a combination of simulation steps (manually executing a part of the code for the first time), and
2. the amount of work needed to implement a change in plans (undoing and doing something else instead).

In UUhistle, the second form of viscosity is unproblematic for a familiar reason: instant feedback means that the user will not embark on lengthy misconceived trips that need to be undone. However, the first form is something of a problem.

Some aspects of execution that are often thought of as a single action require multiple repetitive simulation steps to complete in UUhistle. One of the clearest examples is parameter passing, which requires processing each parameter separately. When the concepts are understood, this is unnecessary repetition. However, requiring the user to deal with each parameter separately is meant to encourage the learner to pay attention to exactly what parameters are involved. In my experience, even if a novice programmer understands how a particular parameter works in a particular case, that knowledge does not necessarily generalize well. For instance, a novice may expect that a particular parameter variable does not need to be created if a variable with the same name already exists somewhere else in memory. For such reasons, and to keep the user interface simpler and more consistent, the increase in viscosity is (arguably) acceptable.

15.4.3 UUhistle's design follows many accepted rules of thumb

Our design and analysis of UUhistle have been informed by various design rules of thumb suggested in the literature. These include Nielsen's well-known usability heuristics (Nielsen, n.d.), Tognazzini's (2003) principles of interaction design, Gloor's (1998) "ten commandments" of educational software visualization design (see also Naps et al., 2003), and cognitive load effects on multimedia learning (Plass et al., 2010; Mayer, 2005, 2009).

There is considerable overlap between these recommendations and the cognitive dimensions discussed above. I will comment on a few salient points that have not yet come up in this chapter.

Adaptation and anticipation

UUhistle shows only those memory areas and components that are relevant to the task at hand (Section 13.1.4). For instance, the call stack is only shown when needed. Given a suitable selection of example programs, this helps the user to make gradual progress and improves the learnability of the system. As programs grow more complex, the initial scaffolding is shed and more features appear in the visualization. Used in this way, UUhistle adapts to the user's growing familiarity with programming and UUhistle itself.

A system that brings the information that is presently required to the user is easier to use than one in which the user has to go and find that information. UUhistle attempts to anticipate the user's needs in many small but – we hope – significant ways in the Info box (see the various figures in Chapter 13). Some of the texts and links that appear there have been tailored to match very specific situations. We are working towards making UUhistle increasingly sensitive to context and a better guesser of user needs.

Abstract graphics vs. standard widgets

Regular visual debuggers and many educational visualization tools from represent program state using standard widgets. In Dönmez and İnceoğlu's VPS system (p. 179 above), too, the learner uses standard widgets for controlling the program.

Following platform conventions and using standard widgets are commonly accepted as parts of good user interface design. UUhistle, however, uses unconventional components both for visualizing state and for program simulation. There are a few reasons behind this approach.

One reason for using tailor-made abstract graphics instead of a 'debugger-like' representation is that we wanted the user interface to reflect the change in perspective that the CS1 student undergoes as they start viewing programs like a programmer rather than an end user. We wished to subtly emphasize to the learner that what they see in UUhistle is a diagram of concepts in a new discipline and represents

something that resides 'within the program', rather than being 'just an interface to an application' in the familiar sense. The separation between the standard widgets used on the left-hand side of the UUhistle window and the abstract graphics that represent memory on the right reflects this distinction (in principle but possibly ineffectually).

Users come in with their existing mental models of standard widgets, such as lists, combo boxes, and radio buttons. This is usually a good thing. However, it is difficult to gauge the impact of those mental models on how the student would apperceive a notional machine visualized using standard widgets. By using an unfamiliar sort of representation for an unfamiliar kind of content we may have avoided some pitfalls at the cost of not being able to fully leverage students' prior UI knowledge.

An obvious benefit of custom graphics is flexibility both in terms of visualization and how the user interacts with them. Another reason we had for abstract 2D graphics was the reported success of the well-researched Jeliot 3 system (p. 165 above), which uses similar graphics for a similar purpose.

Even given the decision to visualize memory as abstract 2D graphics, we might have opted for (and did indeed consider) using a separate set of standard widgets to control the visualization during VPS. However, we decided not to do so for a number of reasons. Most importantly, we wanted the VPSing learner to interact directly with the visualization so that they cannot ignore the visual representation of state and will learn to make use of it. Space concerns were another reason; a separate set of controlling widgets would be hard to fit onscreen, given our quest for maximal visibility (see above). Other reasons concerned specific components that we might have used: buttons, input dialogs, radio buttons, and the like. We wanted each individual simulation step to be quick and convenient to perform once you know what to do. For this purpose, we outlawed all typing in of values and identifiers. Selecting each simulation step from a complex menu or a (long) list of multiple-choice options requires a lot of reading on the user's part and is not convenient in the long run.

In the end, contextual popup menus are effectively the only standard GUI widget that the UUhistle user manipulates during VPS. The popups do not interfere with the visualization of memory and do not take up space.

The clear downside of our approach is that users do not know from the start how they can use the custom components. It can be tricky to know at the beginning where to click to create a frame, for instance. However, because of the consistency of the GUI, there should not be too much to learn.

Cognitive load effects in multimedia learning

Cognitive load theory (Section 4.5) suggests that many 'effects' come into play as learners deal with material that stretches their cognitive capacity. Two of these are the split-attention effect and the modality effect (see, e.g., Plass et al., 2010; Mayer, 2005, 2009). The *split-attention effect* means that learning is hindered when the learner has to keep a part of the learning material in mind while consuming a separate part; a special case of this effect occurs in multimedia learning when the visual channel is overloaded by the need to follow an animation and a textual explanation of it which are spatially separated. The closely related *modality effect* suggests that presenting materials using both the visual and the auditory channels simultaneously is more effective than presenting all the material through only a single channel, especially when the material is difficult or unfamiliar.

One implication of the split-attention effect on VPS design is that support for novice learners should be fully integrated into the environment in which the learning task is carried out, rather than existing as a separate document that needs to be accessed and mentally integrated with the environment by the learner (van Merriënboer and Kirschner, 2007). Otherwise, the support, however useful, is likely to be ignored by the learners, who have enough on their plate without accessing external resources. Through the "What is this?" links and the Info box (see Section 13.1), we have sought to integrate UUhistle's documentation into the visualization environment itself for maximal ease of reference.[9]

A possible concern related to the split-attention effect in UUhistle is that UUhistle's Info box is located outside the graphical visualization of memory, which means that in order to keep track of both texts and graphics, visual scanning is needed. The user also has to keep track of program code, which is in yet

[9] In contrast, in an earlier version of UUhistle, instructions and explanations only appeared at the beginning of each animation; see Section 16.1.

another panel. A better solution in the future might bring explanatory texts closer to where the VPS action is.

Another limitation of UUhistle that is highlighted by the split-attention and modality effects is its lack of support for the auditory channel. The human capacity to receive sensory information is very limited, and one of the few known ways to alleviate the problem is to use visuals and sound simultaneously. The current failure of UUhistle to do so must be viewed as something to improve upon.

The overall effects of UUhistle's user interface on cognitive load are unknown at present.

Conclusions on usability

UUhistle's usability appears fairly good.

The analysis of cognitive dimensions suggests that the design of UUhistle's VPS exercises suits their intended purpose. The laborsome, viscous nature of some interactions is perhaps UUhistle's greatest usability challenge from the cognitive dimensions point of view. The split-attention effect is another cause for concern. There are many small problems that we are aware of and UUhistle is, of course, still a work in progress.

Ultimately, UUhistle's usability depends on how learners use it. We have conducted no formal usability studies. However, thousands of students have succeeded in using the system and most have found it easy enough to use. I present some results from an opinion survey in Chapter 20.

An external evaluation

This chapter has presented our own analysis of UUhistle. In a recent independent review of visualization systems, Gondi (2011) evaluated a visualization example in UUhistle against nine best practices identified in the literature. A 2–3–4-tree visualization in UUhistle was found to support eight of these practices, namely, support for flexible execution control, providing a context for users to interpret the visualization (such as a textual description), providing multiple views or representations, allowing user-specified data sets, showing the source code of what is being visualized, suitability for use as a lecture aid, suitability for self-study, and suitability for debugging.

The ninth best practice, having students answer questions or make predictions, was apparently not present in the particular example evaluated by Gondi. As the reader will know by now, UUhistle does, of course, very much support such modes of interaction as well.

In Part IV, I have presented visual program simulation and a supporting piece of software as the outcomes of what I hope appears to be a rational and logical process.

In a keynote address at the Koli Calling 2010 conference on computing education research, Michael Kölling[10] remarked that the design of pedagogical software by computing educators is often informed by gut feeling, but that this is not bad – gut feelings are important and useful. Nevertheless, Kölling reminded us, with reference to Parnas's famous article *A Rational Design Process: How and Why to Fake It*, it is worthwhile to rationalize our design process and to relate it to theory.

According to Parnas, no software project, past, present, or future, proceeds in the 'rational' way, but we can still make an effort to approach rationality and to document our software as if we had followed the ideal process: the documentation should help the reader to understand what we have created, not to relive the creation process. "It is very hard to be a rational designer; even faking that process is quite difficult. However, the result is a product that can be understood, maintained, and reused." (Parnas and Clements, 1986, p. 256)

Parnas wrote about designing and documenting software artifacts and their modules. My concern in this thesis is not one of software design in Parnas's sense, but the design and documentation of a software-supported pedagogical approach. This work on visual program simulation has not followed a rational, well defined sequence of steps, although this thesis may sometimes portray it as such. The work has been informed throughout by a complex interweaving of gut feeling, literature, and emerging empirical

[10]Say this five times fast: collected colleagues of Kölling at Koli Calling.

evidence. Like the documentation of Parnas's software projects, I hope that Part IV serves as a rationale for VPS to the benefit of anyone who wishes to understand it, use it, or improve on it.

Now it is time to look at some empirical findings.

Part V

Empirical Investigations of Visual Program Simulation

Introduction to Part V

Reflecting on their lengthy experience of developing and researching the Jeliot program visualization system for introductory programming education, Myller and Bednarik (2006, p. 41) wrote:

> *Classroom studies inform* [us] *about the practices taking place* [...] *and can generate testable hypotheses. Controlled experiments, when designed well, can provide answers to the previously established hypotheses and can give accurate insights into interaction and cognitive processes involved in programming. Data from surveys and questionnaire studies can be used both to collect data related to attitudes and current practices, and generate testable hypotheses. Furthermore, all these methods can indicate issues for further development in the form of usability problems or unexpected behavior of users.* [...] *Each of the described methodologies has its own place in the research and development cycle.* [...] *We think that both short-term and longitudinal studies are needed as well as quantitative, qualitative and especially mixed methods studies.*

This part of the thesis contains multiple interrelated studies in which my colleagues and I investigate VPS from many different perspectives. Chapter 16 is a general introduction to the chapters that follow. It reviews our research questions and describes our pragmatic, mixed-methods approach to empirical research and the context of our research. Chapter 17 reports a phenomenographic study of how students perceive learning through VPS. In Chapter 18, we qualitatively explore what happens during VPS sessions. Chapter 19 reports an experimental study in which we quantify the short-term effects of a VPS session on students' program-reading ability. Chapter 20 reviews student feedback on VPS and UUhistle. Together, these chapters constitute a preliminary empirical evaluation of VPS and our implementation of it.

Chapter 16

We Investigated If, When, and How Visual Program Simulation Works

This chapter introduces the interrelated empirical investigations which we will present in detail in the following chapters. It consists of five sections. Section 16.1 is a general commentary on pedagogical research that serves as a background to our choice of research questions. Section 16.2 starts with a recap of the research questions; I then explain how we have sought to address these questions with a pragmatic mixed-methods approach. In Section 16.3, I consider issues of research quality and trustworthiness. Section 16.4 gives some concrete details about our primary research setting – a particular CS1 – and the prototype of UUhistle that students used during the 2010 course offering that we investigated. Finally, Section 16.5 explains who "we" are.

16.1 Evaluations of pedagogy must go deeper than "yes" and "no"

Not every new pedagogy or piece of educational software should be adopted by everyone, even if it is 'a good one'. Context matters. Learners matter. Teacher preferences matter. Institutional strategies matter. It is impossible to incorporate all good things into a single teaching context. Conducting research that informs complex, context-dependent decision making on pedagogy is anything but trivial.

Difficult dissemination

In formal education, teachers have the deciding role in choosing which pedagogical advances and new educational systems to adopt. However, many teachers do not adopt these innovations, even when they and their students might benefit from them. This general concern has also been observed in the specific context of educational software visualization.

According to Hundhausen et al. (2002) and Naps et al. (2003), teachers' reasons for not adopting software visualization tools include the lack of teacher time for learning about systems or configuring them, concern about SV taking away class time needed for other activities, and suspicions of limited educational effectiveness. Ben-Bassat Levy and Ben-Ari (2007) wrote a paper – *We Work So Hard and They Don't Use It* – exploring the reasons behind teachers' reluctance to adopt the unusually well-researched program visualization tool Jeliot. They found that teachers' personal pedagogical styles significantly affected their decisions regarding program visualization and that integrating tools into other learning materials could encourage some teachers to adopt Jeliot.[1] Kaila (2008) interviewed Finnish teachers of introductory programming about their use of software tools, and reports that teachers chose not to adopt tools for a variety of reasons. These included lack of teacher time, excessive overhead in terms of class time, poor fit with specific context (e.g., programming language), the poor usability of tools, the sentiment that tools complicate topics that are already complicated, and issues with the specific visualizations used.

A number of recent initiatives have sought to aid the dissemination of pedagogical practices within computing education in general and educational software visualization in particular. Forums have been

[1] When it comes to software tools for computing education, some of the biggest success stories involve the integration of a tool with a textbook (e.g., Barnes and Kölling, 2006).

launched for sharing (e.g., Fincher et al., n.d.; Korhonen, n.d.; AlgoViz, n.d.), teachers have been given financial aid to lower the collaboration threshold (e.g., Korhonen, n.d.), and technical solutions have been developed to increase intercompatibility and extensibility so that tools are easier to integrate into different contexts (e.g., Moreno, 2005; Karavirta, n.d.).

Such initiatives may alleviate some of the difficulties. Other issues – those that involve teachers' knowledge of and trust in new pedagogies – can be addressed through empirical research. However, one is then confronted with the problem that new pedagogies always work.

Doomed to 'succeed'

In CER, a fledgling discipline, there is often no experiment or any other kind of scientific research setup behind claims of success. A lot of the CER literature consists of so-called Marco Polo papers (Valentine, 2004) that conclude something to the effect of "I did X. It was exciting, seemed to work, and my students liked it". Marco Polo papers are valuable in community building and as inspiration, but do not amount to rigorous evaluation.

Even rigorous evaluations are not always very informative. According to a chagrined sentiment that gets bandied about in educational circles, "every educational experiment is doomed to succeed".[2] Teacher-researchers develop new pedagogies inspired by their particular contexts. They apply those pedagogies enthusiastically themselves, and evaluate them in the context of their own teaching. Not surprisingly, the results of these evaluations are often positive in the sense that the new pedagogy is found to be effective.

Descriptive evaluations needed

As noted, teachers are concerned about the effectiveness of candidate pedagogies, the overhead of adopting them, and their fit with context. To help teachers make well-informed decisions, we need evaluations whose conclusions reach deeper than "yes, this works, you should do it too" (and the occasional "no, this doesn't work"). We as researchers also need to explore how and under what circumstances a new pedagogy or tool works, and precisely what it is good for. Conversely, we must be critical and consider what, how, and when the new pedagogy does not work, and what effort it takes to apply it successfully. There is often a price to pay for benefits gained.

Given a rich set of empirical research findings, the teacher can consider what the new pedagogy could bring to their teaching and whether it would be worth the bother of adopting it. The teacher may weigh pros against cons against the backdrop of their particular context of teaching and learning. Explorative evaluations that go beyond "yes" and "no" can also help the teacher understand the pedagogy and to work out how best to put it into practice (and how not to).[3]

16.2 We used a mix of approaches to answer our research questions

Through the empirical research presented in this thesis, we wished not only to obtain tentative evidence regarding whether visual program simulation is a useful pedagogical approach, but also to be able to describe its good and bad sides, and to discuss what it takes to make use of VPS in teaching. The two overarching questions we wished to answer were:

> *In what ways does visual program simulation impact on learning?*

and

> *How can we improve our implementation of visual program simulation in the UUhistle system?*

[2] This expression is sometimes credited to Albert Shanker – see, e.g., Shanker (1969).

[3] None of this will happen, of course, if the teacher does not have enough time for planning their teaching or following educational research. This significant practical concern I will gloss over here.

We operationalized parts of these broad, related questions as smaller subquestions.

1. In what ways do novice programmers experience learning through visual program simulation?
2. What happens during visual program simulation sessions? In particular:
   (a) In what ways do students justify their choice of simulation steps?
   (b) What other interesting episodes can we observe?
3. Does a short VPS session help produce short-term improvement in learners' ability to predict the behavior of given programs?
   (a) Does a short session of studying examples using UUhistle (v0.2) bring about greater short-term improvement in students' ability to predict program output than studying examples without a visualization does?
   (b) Are there differences in the effectiveness of this treatment for different content?
4. How do students react to the use of UUhistle in CS1 (and why)?

This is a varied bunch of research questions that is best answered using a mix of research methods. Let us first consider some general points concerning mixed-methods research before returning to our research questions and how we attacked them.

16.2.1 Mixed-methods research is demanding but worthwhile

> *Try to imagine a real-life situation that is important to you in which you had to make an evidence-informed decision. What reason could you have for ignoring relevant evidence simply because it was numeric or textual?* (Gorard, 2010, p. 249, parentheses removed)

We adopt an eclectic and pluralistic view of learning theory (cf. Chapter 8). This eclecticism extends to research design and is evident in the empirical work presented in the following chapters. We eschew the extreme positions on qualitative and quantitative research, and attempt to find a middle road.

Our empirical investigations form an interconnected whole that constitutes a body of mixed-methods research. Mixed-methods researchers reject the "incompatibility thesis" that pens qualitative and quantitative purists in their own enclosures, and are open to using both kinds of research as situations call for them (Tashakkori and Teddlie, 2010). We believe that both qualitative and quantitative analyses can contribute to an increased understanding of how visual program simulation impacts on learning, and that both can help us determine when and how it is best used.

Some writers consider mixed-methods research to be a separate research paradigm of its own alongside qualitative and quantitative research. Adopting this view for convenience in this chapter – even though it is an oversimplification (Tashakkori and Teddlie, 2010) – let us (briefly!) consider some of the merits and weaknesses of the three paradigms as they are commonly conceived. The following is largely based on Johnson and Onwuegbuzie (2004) and Patton (2002); many similar lists appear elsewhere in the literature.

Qualitative methods work well for data-driven, naturalistic exploration and the generation of tentative theories and hypotheses. They help us develop rich understandings of phenomena in context and to describe those phenomena vividly, drawing on the meanings that participants attach to phenomena. Qualitative exploration can adapt to what is discovered during the research and can make use of the researcher as a research instrument. The weaknesses of qualitative methods include their relative unsuitability for testing hypotheses, the context-dependence of results, the often time-consuming nature of qualitative data analysis, and a higher risk of researcher bias affecting the findings.

Quantitative methods are particularly useful for testing existing hypotheses and for studying large groups. In a controlled setting, the effect of confounding variables can be minimized to focus on a particular aspect and examine cause-and-effect relationships in a credible way. Numerical data is often relatively quick to process and can be presented precisely and succinctly. The results of statistical analyses are

relatively independent of the researcher. The weaknesses of quantitative methods include an increased risk of missing out on important aspects of a complex situation because they were not covered by hypotheses, a greater reliance on the scientist's views of a situation at the expense of the participants', a tendency to provide highly abstract results that may be difficult to apply directly in specific situations, and a sometimes excessive focus on universal or near-universal generalizability at the expense of context-sensitivity.

Mixed-methods research seeks to use both qualitative and quantitative methods in a way that allows their strengths to complement each other and compensate for the weaknesses of each purist paradigm. Rich qualitative descriptions add meaning to numbers obtained through quantitative work. Numbers add precision to words, pictures, and narrative. The mixed-methods paradigm allows the use of a wide range of research methods, which enables the researcher to pick methods freely on the basis of research questions and their subquestions instead of following the dogma of qualitative or quantitative purists. The weaknesses of mixed-methods research are primarily practical. It takes more time (or more researchers) to learn to use different methods well and to apply them. Similarly, more is demanded of peer evaluators of mixed-methods research. Another practical concern is that the method-mixing researcher runs the risk of offending both qualitative and quantitative purists at the same time.

16.2.2 A pragmatist philosophy of science supports method-mixing

In this thesis, I adopt a pragmatist view of scientific research. In this, I am influenced especially by the somewhat different but related positions of Johnson and Onwuegbuzie (2004) and Phillips and Burbules (2000). These authors draw in turn on the work of classical pragmatists, especially the triumvirate of Charles Sanders Peirce, William James and John Dewey; the pragmatism-influenced postpositivism[4] of Phillips (Phillips and Burbules, 2000) also owes a great deal to the work of Karl Popper.

Johnson and Onwuegbuzie argue that pragmatism is a suitable philosophical partner for present-day mixed-methods research. The following is a list of some of the general characteristics of pragmatism as they define it (selected and adapted from Johnson and Onwuegbuzie, 2004).

Pragmatism...

... has attempted to find a workable solution (sometimes including outright rejection) to many long-standing philosophical dualisms (e.g., rationalism vs. empiricism, realism vs. antirealism, free will vs. determinism, Platonic appearance vs. reality, facts vs. values, subjectivism vs. objectivism) about which agreement has not been historically forthcoming;

... recognizes the existence and importance of the natural or physical world as well as the emergent social and psychological world that includes language, culture, human institutions, and subjective thoughts;

... views knowledge as being both constructed and based on the reality of the world we experience and live in;

... endorses fallibilism (current beliefs and research conclusions are rarely, if ever, viewed as perfect, certain, or absolute). Capital "T" Truth (i.e., absolute Truth) is what will be the "final opinion", perhaps at the end of history. Lowercase "t" truths (i.e., the instrumental and provisional truths that we obtain and live by in the meantime) are given through experience and experimenting;

... views theories instrumentally (they become true and they are true to different degrees on the basis of how well they currently work; workability is judged especially on the criteria of predictability and applicability);

[4] "Postpositivism" is defined very differently by different authors, and I cannot identify my views in many of the definitions (e.g., that provided by Lincoln et al., 2011). Like "positivism", the term is often strongly associated with correspondence theories of truth, quantitative research, and the falsification of hypotheses, which is why it bears stressing that Phillips's postpositivism – which I am partial to – is quite open to various forms of research and reasoning. Phillips's view is related to pragmatism (cf. Phillips, 1975) and is in many ways compatible with it. Of course, "pragmatism" is hardly a single unambiguous position, either.

- ... endorses eclecticism and pluralism (e.g., different, even conflicting, theories and perspectives can be useful; observation, experience, and experiments are all useful ways to gain an understanding of people and the world);
- ... endorses a strong and practical "empiricism" as the path to determining what works;
- ... prefers action to philosophizing;
- ... endorses theory that informs effective practice.

I will elaborate on my position regarding a few related points.

On science, knowledge, and truth

Niiniluoto (e.g., 1980, 2002) – whose writings form the basis of the thesis-writing advice given to doctoral students in Finnish technical universities (Airila and Pekkanen, 2002) – defines scientific research as the systematic and rational pursuit of scientific knowledge. According to Niiniluoto (who builds on Popper), scientific knowledge should be true or, failing that, *truthlike* – a term that means, informally, 'similar to or approximate to truth'. These definitions of scientific research and knowledge rely on a definition of truth.

The critical scientific realism advanced by Niiniluoto is founded on a correspondence theory of truth in which truth is a relationship between reality and meanings expressed through language. It dismisses pragmatistic theories of truth – summarized as "true is that which works in practice" – on the grounds that being useful in practice is less a suitable definition of truth than an emergent property of acting on propositions that correspond to reality.

Truth may indeed not be best defined as what works. It has been proposed that the pragmatist may skip past this objection by treating it as a terminological quibble and conceding it. Dewey warned against identifying knowledge with reality, and suggested that instead of truth we would be better off speaking about "warranted assertibility" (see, e.g., Phillips and Burbules, 2000, p. 3). The scientific enterprise, in this view, is better characterized as the rational and systematic pursuit of warranted beliefs backed up by evidence – empirical and otherwise – but not solidly founded on any single basis. Although absolute Truth is usually, or always, unattainable, the concept of truth can nevertheless be useful as a "regulative ideal" (Phillips and Burbules, 2000, after Popper) towards which scientists strive as they seek to muster ever better warrants for propositions that are useful for acting upon, and probably truthlike, but always subject to revision.

Pragmatists accept that science produces some results that appear to be useful in practice, but are not actually True. Pragmatist science has perhaps an increased risk of failing to discover truthlike knowledge that serves no immediately obvious practical purpose. Despite these weaknesses, I consider pragmatism a 'good enough' and indeed appropriate stance for the purposes of this project. I am satisfied to say that our results lead to warranted beliefs rather than truths – however, the likelihood of an empirically warranted belief being truthlike is better than that of most beliefs. Should our warrants for making our claims be insufficient, I trust the scientific community will do its best to point this out. In the event that our claims are not truthlike enough to be useful, I trust that future research and endeavors to apply the claims will fail to corroborate them, at which point they can be revisited and replaced with improved ones.

Finally, pragmatism stresses that many knowledge claims are highly context-dependent; what can be reasonably asserted about one context can not be asserted about other contexts without powerful warrants for doing so. Pragmatism is a form of practical fallibilism: we can never fully trust our knowledge to work in the novel situations that confront us in our ever-changing world (see, e.g., Biesta and Burbules, 2003).

On values

Qualitative methods purists (e.g., Lincoln and Guba, 1985) contend that value-neutral inquiry is impossible, and, more controversially, that researchers should be openly ideological and allow their values to (openly, not covertly) affect the way they conduct science. Many advocates of quantitative methods agree that absolute value-neutrality is impossible but nevertheless see it as a desirable goal in principle.

Phillips and Burbules (2000) draw attention to a useful distinction made by earlier writers between internal and external values, which matches how we think of values within the present project.

There are values that are *internal* to science and *epistemically relevant* (Phillips and Burbules, 2000, p. 54). For instance, one must not falsify evidence and must be open to criticism. Without these values – which "foster the epistemic concerns of science as an enterprise that produces competent warrants for knowledge claims" – "scientific inquiry loses its point".

External and *epistemically irrelevant* values range from the personal (e.g., "computing education is important and we must make sure everyone who does our courses learns at least the basics") to externally given agendas (e.g., "we need to find out how best to advertise our department to potential future students"). These values may and do (and must) guide the scientist as they choose which research questions are important enough to merit investigation – a decision which may change during an investigation. They also affect the courses of action that we take and recommend on the basis of our results. However, we should minimize the intrusion of epistemically irrelevant values into the process of establishing empirical evidence for knowledge claims about our object of research. In other words, our results are ideally not affected by what we would prefer them to be.

Phillips and Burbules (2000) are careful to point out that "of course the judgment about what *is* epistemically relevant or irrelevant is itself – like *all* judgments – potentially a fallible one" (p. 54).

On methods and results

Various pragmatists and mixed-methods scholars (recently, e.g., Patton, 2002; Johnson and Onwuegbuzie, 2004; Morgan, 2007; Tashakkori and Teddlie, 2010) have argued that there is a disconnect or only a very loose connection between philosophy and research methods. Johnson and Onwuegbuzie (2004), for example, caution that "there is rarely entailment from epistemology to methodology" (p. 15). Patton (2002) makes a similar comment and concludes: "In short, in real-world practice, methods can be separated from the epistemology out of which they have emerged" (p. 136). Similarly, a set of research findings can be useful to researchers and practitioners who subscribe to different ontological and epistemological assumptions than the original researchers. The final word on the epistemological status of our results lies with our readers, who will interpret our results on the basis of their own beliefs.

I will return to the fundamentals of qualitative and quantitative research shortly to discuss the trustworthiness of our research. Now, let us return to our research questions.

16.2.3 The following chapters describe several interrelated studies

The pragmatist may use any methods for empirical work that produce useful results and that suit the research goals. That is what we have done.

To answer Question 1, "In what ways do novice programmers experience learning through visual program simulation?", we adopted a phenomenographic perspective. As discussed in Chapter 7, phenomenography is a research approach that is geared towards studying the different ways in which people experience phenomena in educational contexts. Compatibly with our research questions – and typically of phenomenographic studies – we used qualitative methods to explore learners' understandings of VPS.

We also adopted a qualitative orientation to answer Question 2, "What happens during visual program simulation sessions?" More specifically, we used a form of data-driven, qualitative content analysis, an approach suitable for the exploration of complex data.

We approached Question 3, "Does a short VPS session help produce short-term improvement in learners' ability to predict the behavior of given programs?" quantitatively, and used an experimental setup to measure the short-term impact of VPS in classroom use.

Finally, we sought to answer Question 4, "How do students react to the use of UUhistle in CS1 (and why)?", through both qualitative and quantitative analyses of course feedback questionnaires, which are an inexpensive way of surveying the opinions of many students.

Table 16.1 presents an overview of our studies.

Table 16.1: An overview of the empirical studies in Part V. The way we have broken down each study into a research purpose, framework, data source, and form of analysis is adapted from Malmi et al. (2010). Only the primary research questions, purposes, and methods are shown.

Chapter	Research question	Research purpose	Research framework	Data source	Form of analysis
17	In what ways do novice programmers experience learning through visual program simulation?	qualitative evaluation, formulating a model	phenomenography	interviews	qualitative analysis
18	What happens during visual program simulation sessions?	qualitative evaluation	content analysis	observations, interviews	qualitative analysis
19	Does a short VPS session help produce short-term improvement in learners' ability to predict the behavior of given programs?	quantitative evaluation	controlled experiment	pre- and post-test	statistical analysis
20	How do students react to the use of UUhistle in CS1 (and why)?	quantitative and qualitative evaluation	survey	online questionnaire	descriptive statistics, qualitative analysis

16.3 You decide if you trust our research

> *Don't label* [your own inquiry processes and procedures] *as 'objective', 'subjective', 'trustworthy', 'neutral', 'authentic', or 'artistic'. Describe them, and what you bring to them, and then let the reader be persuaded, or not, by the intellectual and methodological rigour, meaningfulness, value and utility of your result.* (Patton, 2002, p. 576)

To establish 'truth' – that is, warranted assertibility – we need good research. What does this mean for our pragmatic mixed-methods project? The qualitative and quantitative paradigms put forward different quality criteria but also share some common ground. Below, I deliberately use stereotypes of polarized purist positions in each camp as I discuss the two paradigms.

16.3.1 Quantitative and qualitative research share the goals of authenticity, precision, impartiality, and portability

Within the quantitative paradigm, research is traditionally assessed in terms of its validity and reliability. Some qualitative researchers and mixed-methods researchers use these same terms albeit often with adjusted definitions – something that quantitative purists jealously guard against. Some qualitative purists instead reject the terms as oppressive relics of the quantitative regime. In a seminal account, Lincoln and Guba (1985) presented credibility, transferability, dependability, and confirmability as the qualitative researcher's alternatives to traditional criteria for trustworthiness.

Onwuegbuzie and Johnson (2006) argue that because of such tensions, using terms such as validity in mixed-methods research is counterproductive. They suggest *legitimation* as a neutral term to be used instead when discussing the overall criteria for the assessment of mixed-methods studies.

The legitimative concerns of qualitative and quantitative researchers are similar at heart. Both wish to perform good research and to convince readers to pay attention to the results reported and find value in them. Both agree that to be scientific, a rigorous, rules-based, systematic approach to research is required. Wesley (2011) builds on the more confrontational work of Lincoln and Guba (1985) to propose a bridge-building terminology that captures the commonality in the two paradigms. I paraphrase:

- *Authenticity* is an abstraction of validity (or measurement validity) in the quantitative paradigm and credibility in the qualitative paradigm. Research is authentic when it is appropriately geared towards research questions.
- *Precision* is an abstraction of reliability in the quantitative paradigm and dependability in the qualitative paradigm. Research is precise when it has been competently carried out.
- *Impartiality* is an abstraction of objectivity in the quantitative paradigm and confirmability in the qualitative paradigm. Research is impartial when it is free of inappropriate bias.
- *Portability* is an abstraction of generalizability (external validity) in the quantitative paradigm and transferability in the qualitative paradigm. Research is portable when lessons can be taken from it to other contexts.

The two paradigms have different strengths. Authenticity has been viewed as difficult to establish in quantitative research, while qualitative research arguably struggles more with the other three facets of legitimation. Each paradigm can learn something from the other. Below, I use Wesley's terminology as I review what the two paradigms have to say about legitimation and the concrete procedures suggested in the literature for establishing legitimacy. The discussion here draws primarily on Lincoln and Guba (1985), Kvale (1996), Patton (2002), Airila and Pekkanen (2002), Breakwell et al. (2006), and Wesley (2011).

Authenticity

Within the quantitative paradigm, the assessment of authenticity (validity) has at its core the question: are you measuring what you intend to measure? Authenticity arises from good research design, in particular from operationalizing phenomena of interest into variables, using appropriate instruments (tests) for measurement, and designing controlled experiments so that observations about a dependent variable can be attributed to controlled variation in an independent variable. Threats to validity include poor design of tests, sampling bias, testing effects (e.g., pretests affecting ability to do what is being measured), maturation effects and history effects (the gradual or sudden influence of outside factors), research subject dropouts, and interference between control and experimental groups (e.g., experimental treatment spreading to the control group; demoralization of the control group).

Researchers within the qualitative paradigm are likewise concerned with the appropriateness of their research instruments for what they are trying to accomplish. The qualitative researcher is not as focused on measurement, but asks a broader question: are you investigating what you intend to investigate? To be authentic (credible), the researcher must produce a report that is tenable as judged by its readership. Kvale (1996) distinguishes between two kinds of authenticity that qualitative researchers rely on. *Communicative validity* is the extent to which research holds up in critical public discussion among either a community of research experts or members of the population being investigated. *Pragmatic validity* refers to the extent to which the research assists in producing useful actions, which also means that the research must be judged to be authentic enough by decision makers. The authenticity of qualitative research rests on the choice of methods and research instruments (to gather high-quality data and to analyze it systematically), and the measures taken to ensure that the findings are credible to the consumers of the research. Techniques include triangulation, peer evaluation, rich 'thick description' of method, setting, and analysis, and extensive quoting from empirical data. (More on specific techniques below.)

In interpretive qualitative research, the researcher is seen as a key research instrument. Accordingly, Patton (2002) emphasizes that authenticity depends on the researcher's personal attributes, such as training, experience, track record, and status.

Precision

In the quantitative tradition, precision (reliability) is largely defined by how replicable a research effort and its findings are. Assuming that the original researchers have carried out their research plan competently enough, and their instruments are designed in such a way that they produce consistent measurements, a repeat study would be expected to produce the same findings even if carried out by other (competent) researchers.

As noted, the qualitative tradition views the researcher as a key research instrument whose involvement and interpretations are reflected in the research results. Just as different tests may produce different insights into a phenomenon, different researchers may discover different insights as they study a phenomenon – in fact, even the same researcher (who changes with time and deepens their understanding during research) may discover new facets of the object of research with repeated attempts. Precision (dependability) in the qualitative paradigm, too, is concerned with competence, consistency, and the elimination of mistakes. However, instead of requiring findings to be replicable, the qualitative tradition instead mandates that the researcher should demonstrate that their findings are consistent with empirical data and follow procedures that reduce the risk of careless mistakes.

Impartiality

Traditional quantitative science seeks impartiality in the sense of objectivity: objective knowledge is something that is 'Real', does not depend on anyone's opinion, and can be tested independently of any particular researcher. Research questions are put to, and research findings are about, 'Nature Itself'. Measures are taken to ensure that findings are as free of researcher interpretation and as unsoiled by values as possible. The researcher seeks to maintain a distance from the object of research through large sample sizes and quantitative methods.

As with authenticity and precision above, the qualitative researcher's conception of the researcher as a research instrument brings a different perspective on impartiality (confirmability). Although the results of interpretive qualitative research – and indeed any research – depend on the researcher, they can be impartial to the extent that they are free of prejudice. According to Patton (2002, p. 575) distancing oneself from the data "does not guarantee objectivity; it only guarantees distance". Similarly, it is argued that the use of measurement and statistics does not protect against bias, as any research design inevitably involves countless researcher-dependent decisions. Patton, citing Lincoln and Guba, prefers to approach impartiality in the social sciences through the notion of "fairness": the researcher should test for and rule out (or acknowledge) his own biases and seek to allow multiple competing viewpoints fairly and equally.

Portability

The quantitative paradigm entrusts the researcher with the quest of producing portable (externally valid) knowledge that applies widely, perhaps even universally, across settings, people, and time. An important estimate concerning the portability of findings arises out of the sampling model: the researcher must judge to what extent their sample is representative of a more general population. Large sample sizes reduce the probability of chance findings that represent only the sample rather than the general population.

The qualitative paradigm has a very different take on portability (transferability). Many scholars emphasize that knowledge of both the source and target contexts is needed to make estimates about portability. It is neither necessary nor appropriate to hold a researcher responsible for 'proving' the portability of their findings. Assessing portability is primarily a task for the consumer of the research, who knows the target context. The job of the original researcher is to provide sufficient information for the consumer to be able to make informed judgments about the similarity of the two contexts.

16.3.2 The literature presents an assortment of techniques for research legitimation

The four aspects of legitimation are connected in complex ways. Authenticity and portability depend on precision, for instance. Many of the techniques presented in the literature impact on multiple aspects of quality either directly or indirectly. I list here some techniques for increasing and establishing authenticity, precision, impartiality, and portability without going into detail about each technique or linking them to specific aspects of legitimation. (For more information, see, e.g., Lincoln and Guba, 1985; Kvale, 1996; Patton, 2002; Airila and Pekkanen, 2002; Breakwell et al., 2006; Wesley, 2011.)

- *Being clear and explicit in general:* reflecting on and clearly reporting research questions and the motivations behind them; rich, 'thick' description of the research setting and methods and of the analysis process; assessment of methodological choices on the basis of their suitability for the research questions.

- *Triangulation:* triangulation of data sources, methods, research designs, perspectives, and theories (either to show that they produce similar results or at least to check for such similarity and analyze discrepancies); replication studies or continuation studies; team research.
- *Researcher capabilities and learning:* reflection on and publication of personal and professional information of relevance about researchers; prolonged, open-minded researcher engagement with the phenomenon of interest within context; pilot studies; training in the use of instruments and methods.
- *Data collection:* carefully justified choice of data sources and sampling strategies (explicitly aligned with each other and with research questions); large sample sizes; explicit justification of how phenomena have been operationalized as variables; controlling and randomizing variables; construct validity checks (e.g., comparing an instrument to another); reflection on possible pitfalls in the use of the methods chosen.
- *Analysis and analyses:* repeated passes through data; analyses of test consistency (e.g., test-retest checks); computer-aided analysis; intercoder reliability calculations and other team-based analysis techniques; demonstration of how alternative explanations were looked for; explicitly reporting uncertainties in analysis; exploration of discrepant cases in the data that do not fit the generalizations made; "member checks" of findings by research participants or others like them; explicit linking of conclusions to empirical data (e.g., through abundant verbatim quoting of data such as interviews); publication of all data.
- *Peer assessment:* peer review by the research community, formal audits of process.

In the studies that follow, we have used some of these techniques. Others we have not used, due to lack of time, inconvenience, or their unsuitability for the research questions, and because we deemed what we did to be sufficient for present purposes.

In the spirit of the quote from Patton at the beginning of this section (p. 255), we leave it to the reader to reflect on research quality and our take on it, and to judge the quality of our craftsmanship as we present what we did in the chapters that follow.

16.4 Our data comes from a certain CS1 course offering

Our empirical data comes from the course *T-106.1208 Basics of Programming Y (Python)* at Aalto University, and specifically from its offering during the spring 2010 semester. During this course offering, we conducted student interviews (see Chapter 17), video observations (see Chapter 18), and a controlled experiment (see Chapter 19). By way of triangulation, we also review some student feedback from course-end questionnaires (Chapter 20).

Below, we will refer to the course T-106.1208 with the label *CS1–Imp–Pyth*, short for CS1 – imperative programming – Python.

16.4.1 We studied a CS1 for non-majors

CS1–Imp–Pyth is a large-class CS1 course for non-computing majors at a multidisciplinary research university whose mission is to bring together technology, economics, and art and design. Hundreds of students from a variety of scientific and engineering disciplines take CS1–Imp–Pyth each spring term. A total of 759 students enrolled in 2010. For many – probably most – of the students, CS1–Imp–Pyth is the only programming course they ever take.

CS1–Imp–Pyth is designed to teach about fundamental aspects of programming that engineering students are expected to find useful in their future work. The primary focus is on having students practice the writing of small, script-like imperative programs. In terms of content and ordering of topics, the course is a traditional CS1 that starts from variables and basic operators and selection, then turns to iteration and functions. Graphical user interfaces and object-oriented programming are touched upon near the end of the course. The course uses the Python language.

Many students take CS1–Imp–Pyth mainly because it is obligatory for students in their degree program. Problems with student motivation have been reported anecdotally by course staff. Many students find the course very difficult, and struggle to master even the basics. The drop-out rates in past years have oscillated between (roughly) 20% and 40%. Certainly, many students do well also; there is great variation in learner attitude, aptitude, background, and performance.

Assignments

Programming assignments are the main learning activity in CS1–Imp–Pyth. There are two main kinds of assignments: program-writing assignments and program-reading assignments.

In 2010, there were 27 program-writing assignments and 24 program-reading assignments. The assignments were divided into 9 rounds. Each round had a deadline, usually a week after that of the previous round. Students worked on the assignments individually.

The program-writing assignments were fixed in scope (as opposed to open-ended). Each assignment required the student to write a small program that produces console output as specified. The programs were small. The most complicated assignment required writing a program that reads in the chemical formula of a molecule (no parentheses allowed) and prints out the corresponding molecular mass.

The program-reading assignments were implemented in UUhistle, as follows.

The role of UUhistle and VPS

Most rounds contained small program-reading assignments. These included program animations in which students merely watched UUhistle animate a given program's execution, and VPS exercises, in which they had to manually simulate given programs.

On the course web site, the program-reading assignments were listed separately from the program-writing assignments and other course materials; the program-writing assignments and other materials also did not refer to the program-reading assignments . Students did all course assignments in whichever order they chose to. The program-reading assignments were numbered in such a way that each topic was first introduced through an animation and then elaborated on in one or more VPS exercises.

Here are the topics of the program-reading assignments in UUhistle:

Round 1: Variables, arithmetic, and I/O (two animations, five VPS exercises)
Round 2: Selection and boolean logic (one animation, two VPS exercises)
Round 3: `while` loops (one animation, one VPS exercise)
Round 4: Functions and parameters (one animation, three VPS exercises)
Round 5: More functions (two VPS exercises)
Round 6: (no program-reading assignments)
Round 7: (no program-reading assignments)
Round 8: Recursion (one animation, one VPS exercise)
Round 9: Classes, objects, and methods (two animations, two VPS exercises)

The programs appear in Appendix B.

Two short introductory videos were published that introduced students to UUhistle's visualizations and to VPSing. They were linked to from the first UUhistle assignments and from the course web site. (Some students watched them, others did not.) Students did not receive any other guidance in the use of UUhistle unless they asked for it themselves in the voluntary exercise sessions (see below).

Assessment and grading

Students submitted their solutions to assignments into a web-based course management system. The solutions were assessed automatically. Program-writing assignments were assessed by comparing the submitted program's output to that produced by a model solution. UUhistle graded students' solutions to program-reading assignments (see Section 13.5) and sent their scores to the associated courseware.

Each assignment was worth a number of points; the overall assignment grade was determined by the sum of the points received during the semester. The program-writing assignments were by far the most valuable in terms of points. VPS assignments were worth considerably less (as was natural, since they were meant to be easier and quicker to do than the program-writing assignments). Even the program animations were worth a very small number of points that the student got just for watching the animations. In total, the program-reading assignments accounted for 14% of all the assignment points available.

Getting the highest grade for the course required the production of a working solution to most assignments. The overall assignment grade was averaged with the grade from a final examination to produce the overall grade.

Students could freely choose whether to do any of the program-reading assignments in UUhistle. Getting the highest grades required them to do at least some of them, but it was possible to pass the course without ever using UUhistle at all.

Exercise sessions and lectures

All the assignments in CS1–Imp–Pyth – program-reading and program-writing alike – were 'open labs'. Students worked on them at their own pace, mostly outside of class. There were no 'closed labs' with a fixed agenda. Instead, teaching assistants supervised 'exercise sessions'. In these sessions, students worked freely on whichever assignment they wanted, and could ask the assistant for guidance when they felt they needed it. Attendance was voluntary. Students could also voluntarily attend lectures. Most students in CS1–Imp–Pyth did not attend either the lectures or the exercise sessions, preferring to study on their own or with friends.

The lecturer occasionally made use of UUhistle to illustrate program execution during the lectures. The visualizations in UUhistle were the main (and almost only) explicit form of teaching about the underlying execution model of programs in CS1–Imp–Pyth. During one lecture at least, the lecturer drew diagrams of computer memory to illustrate references.

Staff

For a number of recent years, CS1–Imp–Pyth has been designed, lectured and organized by a teacher who has not otherwise been involved in UUhistle's development or this empirical study. She agreed to adopt UUhistle after it was suggested to her during a demonstration of the tool by the authors. The program-reading assignments were created by UUhistle's authors and subsequently approved by the teacher. UUhistle's programmer, Teemu Sirkiä, was one of 18 part-time teaching assistants. He did not play a direct role in conducting the empirical studies presented in the next chapters.

The teaching assistants were asked to familiarize themselves with UUhistle and VPS on their own and did not receive any other training on the topic.

A few statistics

A total of 562 students did at least one of the program-reading assignments. That means 74% of all enrolled students, and 81% of those who submitted at least one assignment of some sort – among the enrolled students there are always some "ghosts" (Mason and Cooper, 2012) who fail to turn up at all or at least do not put in nearly any work.

The students made 11,010 submissions through UUhistle in total, including a small number of resubmissions.

The code of each of the program-reading assignments was fairly short, ranging from 2 to 28 lines in length. The shortest program animation (assignment 1.1, that is, the first program-reading assignment of round 1) was 13 (correct) steps long, and the longest (assignment 9.3) about 200 steps. The shortest VPS exercise (assignment 1.2) was 12 steps, and the longest (assignment 4.9) about 80 steps.

The easiest VPS task as measured by the number of mistakes made was assignment 3.3 on `while` loops, in which the submissions had a median of 1 misstep. The hardest was assignment 5.1 – which combined function calls, boolean values, selection, and logical operators – with a median of 30 incorrect steps per submission.

Students typically spent a few minutes on each VPS assignment. Assignment 1.4 was the VPS assignment with the shortest average completion time; students spent a median time of 1.4 minutes on it. By the same measure, the longest exercise was the above-mentioned assignment 5.1, completed in a median time of 14 minutes.[5]

Language

The course language was Finnish. Finnish identifiers were used in program code. The students used a Finnish-language version of UUhistle. The author of the thesis has translated all the quotes and identifiers in the following chapters from Finnish to English. We are not aware of any significant language-dependence in our results.

16.4.2 In the studies that follow, students used an early prototype of UUhistle

Chapter 13 described UUhistle v0.6, the most recent release at the time of writing. The offering of CS1–Imp–Pyth in Spring 2010 used UUhistle v0.2, an earlier prototype. There are some possibly significant differences between the two versions, which the reader should bear in mind when reading the following chapters.

- The Info box, the explanatory texts that it links to, and the "What is this?" menu choices accessible through context menus (see Section 13.1.3) did not exist in v0.2. A simple textual explanation of the previous execution step (in animations) or a prompt (in VPS exercises) was shown on the status line at the bottom of the window (see Figure 16.1).

- Assigning to a new variable involved two entirely separate steps: first you created an empty variable using a context menu (see Figure 16.1) and then dragged the value to it. (Cf. in UUhistle v0.6 you assign to a new variable by dragging the value to the frame, which brings up a context menu for creating and naming the variable.)

- The Next Line / Fast Forward button did not exist.

- UUhistle v0.2 gave almost no context-specific feedback, nor did it attempt to draw the user's attention to points of interest or directly address specific misconceptions through the textual materials (cf. Sections 13.1.3, 13.4.2, and 15.3). UUhistle notified the user when they made a mistake, but all textual feedback was very generic, along the lines of "You performed the wrong kind of execution step." or "You used the right value but misplaced it.", and the user had to click a button (the one with the question mark icon) to receive it.

- When the user moused over a reference in v0.2, the target object was highlighted in green but no arrows were drawn (cf. Figure 13.1 on p. 194).

- (Many additional features and small tweaks with a lesser impact on the present work.)

The students of CS1–Imp–Pyth used the 'lean build' of UUhistle v0.2 (see Section 13.7), which only serves as a client for ready-made program-reading assignments. That is, students could not edit code in UUhistle or use the system to animate their own code.[6]

We close this chapter with a brief exposition of ourselves.

[5] The figures given here have been calculated from UUhistle's submission logs. The times are measured from opening an assignment to the submission timestamp. Abandoned sessions (which did not result in a submission) are excluded.

[6] The main reason for using the lean build was that we were concerned about technical problems that would emerge if students were to run arbitrary Python code within the early UUhistle prototype.

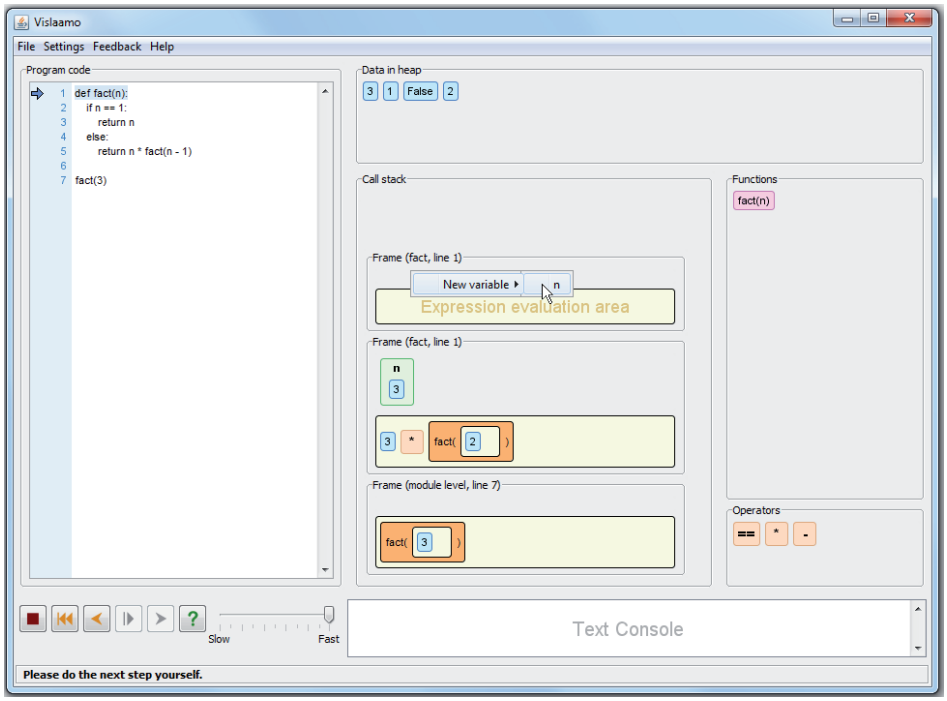


Figure 16.1: The earlier version of UUhistle (v0.2) that was used in the 2010 offering of CS1–Imp–Pyth. Cf. Figure 13.9 on page 203, which shows the newest version (v0.6).

16.5 Who are "we"?

The author of this thesis, Juha Sorva, was the lead researcher in each of the empirical evaluations of VPS in Part V. The thesis supervisor, Lauri Malmi, also had a hand in each of the projects. Jan Lönnberg participated in the work we present in Chapter 17. We will describe the roles of the researchers separately in each chapter.

We undertook this research in our roles as teachers at Aalto University and computing education researchers in the university's Learning+Technology research group.[7] Our goals were to evaluate VPS and UUhistle in order to improve our own teaching, our students' learning, and the teaching and learning of others elsewhere, and to produce this dissertation.

The author of the thesis has many years of experience in teaching introductory programming courses at Aalto. He has made the occasional contribution to the design of materials for CS1–Imp–Pyth but has never taught that course himself. Lauri Malmi is the leader of the Learning+Technology group and the head of introductory programming education at Aalto University, and has long experience within computing education, software visualization, and computing education research. Jan Lönnberg is a doctoral student in computing education research,[8] with teaching experience from a concurrent programming course.

All of us had taken courses in research methods (both quantitative and qualitative, including phenomenography); the thesis supervisor had also given such courses. We all had previously written peer-reviewed publications reporting empirical research unrelated to this thesis. The two doctoral students were nevertheless still fairly inexperienced as researchers.

None of us was involved in the teaching of CS1–Imp–Pyth, but we all know the teacher as a colleague. The thesis supervisor is married to the CS1–Imp–Pyth teacher, but is not directly involved in decision making regarding the pedagogy of that course.

[7] What is now known as the Learning+Technology group was formed when the Software Visualization Group (SVG) and the Computing Education Research group (COMPSER) merged in 2010, during this thesis project.

[8] *Was* at the time, I should say. He beat me to finishing (Lönnberg, 2012).

We must acknowledge the possibility that our attitudes and expectations may have biased our research. We come from a professional subculture within which there exists significant belief in the educational use of visualizations in general and interactive visualizations in particular. Interest in, and the use of, software visualization systems such as those reviewed in Chapter 11 is considerable within our community. We, too, were hopeful that our tool would turn out to be valuable for CS1 teachers and students, including ourselves. As the author of the thesis is one of the designers of UUhistle, there is a risk of bias in favor of results that portray the system in a positive light. As we conducted the research, we actively sought at all stages to be aware of our own hopes and views on VPS, to remain open-minded to others' different views, and to honestly identify problems in the use of VPS (some of which we expected to exist on theoretical grounds; see Section 14.5 and Chapter 15). Such a critical attitude, we feel, is necessary even from a selfish point of view: while we hope to be able to make use of VPS in the future in our own teaching and research if it is a worthy pedagogy, we believe in empirical evaluation as the key to deciding if, when, and how to use VPS.

Chapter 17

Students Perceive Visual Program Simulation in Different Ways

This chapter reports on an empirical study conducted by the author of the thesis with the help of Jan Lönnberg and Lauri Malmi. We investigated the question:

> *In what ways do programming students experience learning through visual program simulation?*

Other studies (e.g., Isohanni and Knobelsdorf, 2010) have shown that programming students often do not use program visualization tools in the ways that their teachers had in mind. Not all ways of using these tools are equally likely to result in effective learning. VPS can only be successful if students relate to it, and use it, in a meaningful way. Taking this as a point of departure, we sought to examine aspects of the relationship between programming students and a software visualization system that supports VPS.

In this study, we do not test existing hypotheses. Our study is *exploratory* and *qualitative*. We seek to expand our knowledge of learning through VPS by finding out, through an analysis of empirical data, what different ways there are of understanding the phenomenon. Our results may serve as a basis for future hypothesis-testing experiments or may be extended by other exploratory studies.

Learning about VPS

The way we formulated our research question arises out of the phenomenographic theory of learning (Chapter 7). In the phenomenographic view, powerful ways of acting on a phenomenon are made possible by rich ways of perceiving the phenomenon.[1] Effective learning through VPS therefore presupposes a sufficiently powerful way of perceiving what learning in a VPS context can be.

By posing and answering our research question, we sought to identify what students need to understand *about VPS* so that VPS will be useful to them for learning programming. Phenomenography posits that the most significant form of learning involves becoming able to experience phenomena in a qualitatively different, richer way than before. It is such critical differences in understanding that we explore in this study. In particular, we wished to discover what obstacles there are to students using VPS effectively – what educationally critical aspects in the phenomenon of learning through VPS a student must discern in order to develop a qualitatively better understanding and make the most of VPS – so that those obstacles can be tackled in teaching.

Chapter structure

Chapter 7 has already introduced some of the fundamentals of phenomenographic research. In the first two sections below, we delve a bit deeper into phenomenography in order to clarify our research question and to review the tools that the phenomenographic tradition provides for answering it. Section 17.1 focuses on theory and Section 17.2 on research-methodological considerations from the literature. In

[1] We use the terms 'way of experiencing', 'way of perceiving', and 'way of understanding' effectively synonymously with each other, in the sense described in Chapter 7.

Section 17.3, we get to a more concrete explanation of what we did in practice to collect and analyze data. Section 17.4 presents the results of our study. In Section 17.5 we discuss what the results mean from the point of view of learning to program. We build on that discussion in Section 17.6 to present some pedagogical recommendations. Finally, in Section 17.7 we acknowledge some vague spots in our findings and some plausible alternative analyses.

17.1 We adopted a phenomenographic perspective on how students perceive learning through VPS

Before we proceed, let us bring to mind some of the main points made about phenomenography in Chapter 7.

- Phenomenography seeks answers to questions of the form "in what ways do people experience phenomenon X?" and subquestions thereof.
- The 'unit of phenomenographic research' is a way of experiencing a phenomenon, defined as an internal two-way relationship between the experiencer and the experienced.
- A phenomenon is constituted by the different ways in which it can be experienced. The question "in what way do people experience a particular phenomenon" does not ask – and phenomenography does not answer – only about learners or the phenomena they experience, but about both at the same time.
- A phenomenographic study investigates the relationship between a population of experiencers and a phenomenon. The researcher attempts to 'see through the eyes of the learners' in order to describe the different ways in which they as a collective experience the phenomenon. The results are the researcher's interpretation of others' experience.
- The typical phenomenographic study produces an *outcome space* of a few *categories of description*. Each category describes a way of understanding the phenomenon of interest that is qualitatively different from the other ways of understanding the phenomenon. (Chapter 7 gives several examples of outcome spaces.)
- The categories in an outcome space typically have logical, hierarchical relationships. Some categories can be said to be richer, more powerful, and more desirable from an educational point of view than others.

In our study (Figure 17.1), the phenomenon is learning through VPS, by which we mean any learning that involves a visual program simulation context (in the students' experience). A way of understanding learning through VPS involves, and relates to each other, a way of perceiving what VPS is, and a way of understanding what one can learn about while doing VPS.

17.1.1 We studied the 'act' and 'why' aspects of learning

Our research question can be made clearer still. What do we mean when we say "learning through VPS"? We can clarify this by analytically viewing learning as several aspects. Marton and Booth (1997, Chapter 5) separate learning into a 'how' aspect and a 'what' aspect (Figure 17.2). The 'how' aspect of learning is the way one learns. It is crucial for the learning outcome as it defines what the learner's goals are and how he goes about achieving them. The 'what' aspect of learning is the content that one learns about, e.g., computer programming or a particular concept. The two aspects are interrelated. How the learner goes about learning affects what he learns about; conversely, the content of learning affects how he goes about learning.

The 'how' of learning can be further divided into the 'act' of learning and the 'why' of learning. The 'act' of learning refers to the activities – of any kind – that the learner engages in as they learn. The 'why' of learning – also called the motive or the indirect object of learning – is what the learner wants to be able to accomplish as a consequence of learning.

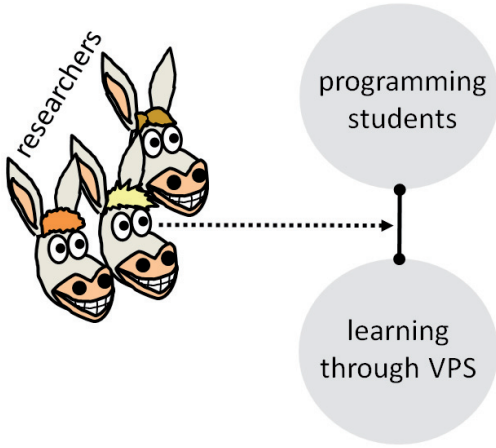


Figure 17.1: The phenomenographic research perspective we adopted in this study. The phenomenon is learning through VPS. The people experiencing the phenomenon are novice programmers taking CS1. Three researchers are studying the relationship between the experiencers and the phenomenon. (This is an instantiation of the general case depicted in Figure 7.1 on p. 95.)

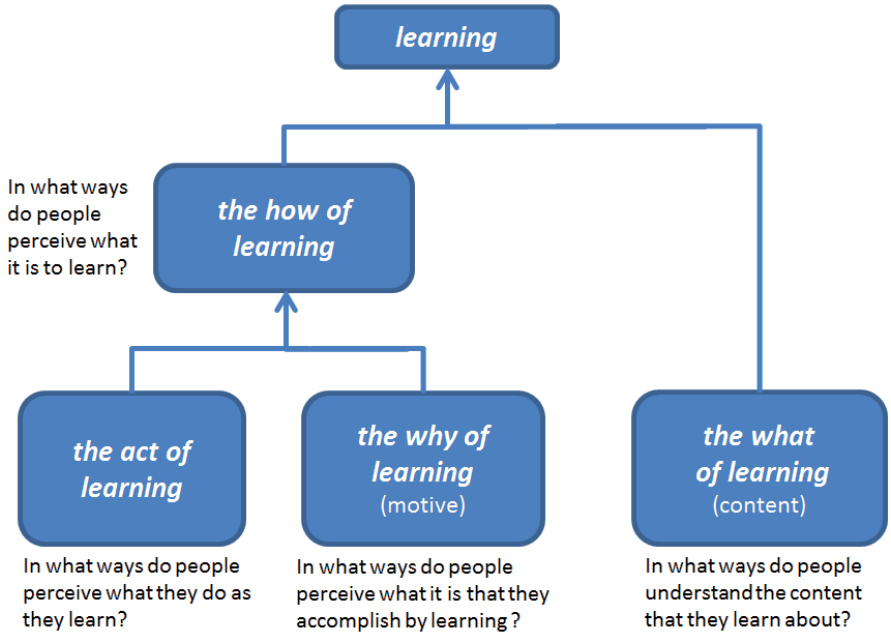


Figure 17.2: Aspects of learning, adapted from Marton and Booth (1997). For each aspect, we have listed a pertinent research question.

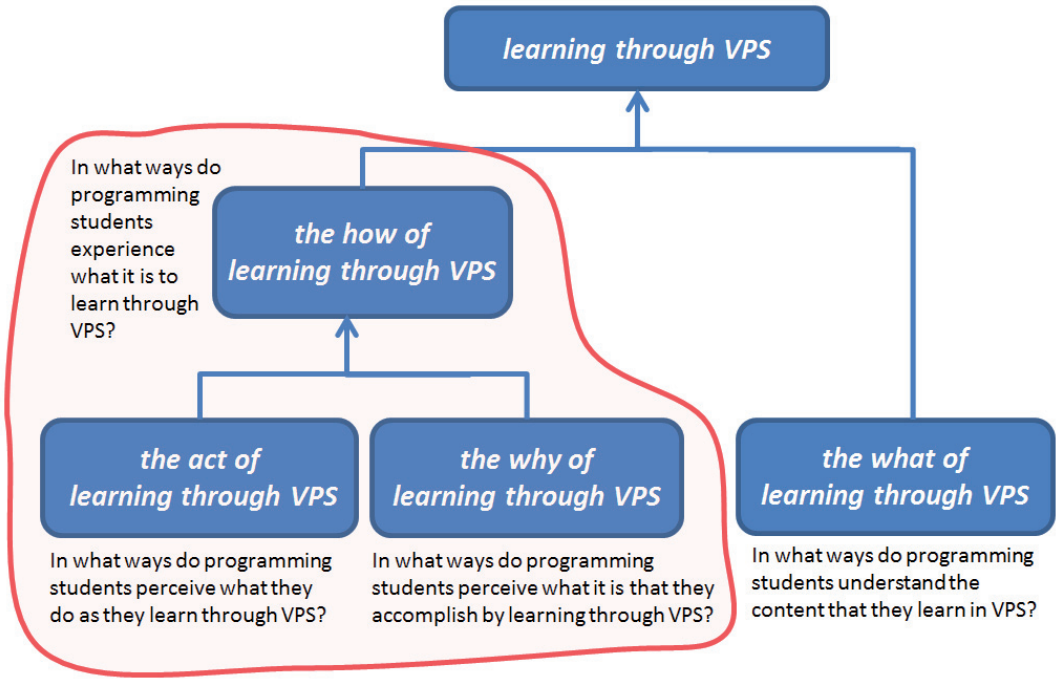


Figure 17.3: Aspects of learning applied to our research on VPS. As in Figure 17.2, we have listed a pertinent research question for each aspect. The research focus of our study is highlighted.

Just as learning can be experienced in qualitatively different ways, so can any of these aspects of learning. In a particular phenomenographic study the researchers may be interested in investigating one or more of these aspects. In our case, we are interested in the 'how' aspect of learning through VPS, as we have illustrated in Figure 17.3. This interest encompasses both the 'act' and the 'why' of learning. In contrast, our research focus is not on the content of learning, the 'what' aspect. We are not seeking answers to questions in the vein of "in what ways do students experience control flow / the execution model of programs / computer memory / the call stack?". In fact, we do not presume to know what content it is that students feel they learn about in a VPS context. What content is perceived as being learned through VPS is one of the things that we hope will emerge through our study of the 'how' aspect of learning.

Marton and Booth (1997) stress that the distinction between the aspects of learning is an analytical one made for the researcher's benefit to distinguish between different research points of view, and that in people's experience these aspects are not separate entities but facets of an undivided whole.

17.1.2 Marton's theory of awareness further structures our research

We have already established that when phenomenographers say 'experience' they do not mean it in just any sense of the word, but in the relational sense described in Chapter 7. Some phenomenographers have found it useful to further dissect the nature of human experience. Of particular interest for present purposes is Ference Marton's theory of awareness.

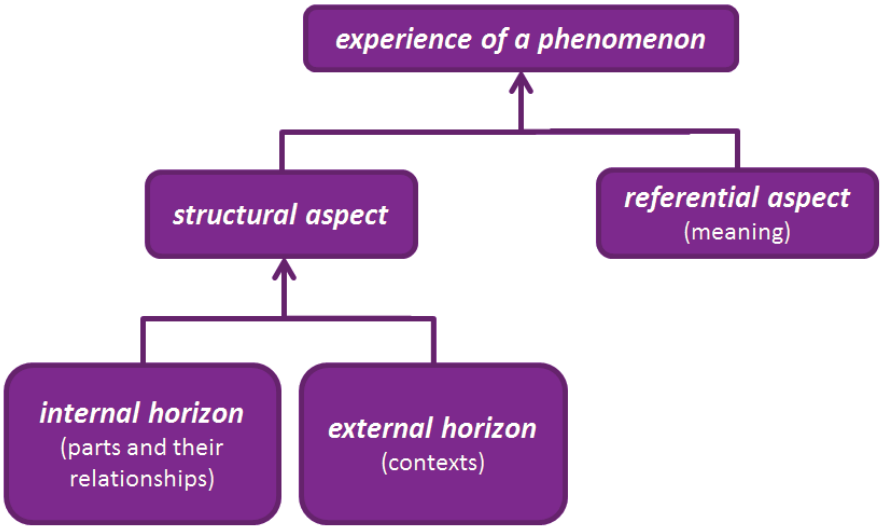


Figure 17.4: The structure of human awareness, adapted from Marton and Booth (1997).

A structure of awareness

> *In a sense we could say that we are aware of everything all of the time. But we are surely not aware of everything in the same way. Certain things are focused, others less and less. Of most things we are only very, very marginally aware. Then the situation may suddenly change and with it the structure of our awareness.* (Marton and Booth, 1997, p. 98)

According to Marton and Booth (1997, Chapter 5), an experience of something can be expressed in terms of the structure that our awareness has at a given time. This structure is represented diagrammatically in Figure 17.4.

To illustrate, Marton and Booth use the example of coming across a deer in dark woods. As we recognize what we see as a deer, we simultaneously experience both a structure and a meaning. The *referential aspect* of an experience is discernment of meaning: what I see is a deer. The *structural aspect* of an experience is the discernment of the structure of the phenomenon. To discern the structural aspect of the deer we must make out its individual parts (legs, head, etc.) and their relationships to each other and the whole deer. Marton and Booth call this the *internal horizon* of the experience. The internal horizon encompasses what is focal to our experience of this phenomenon. To experience the deer *as* a deer, we must also relate it to, and distinguish it from, its surroundings. The dark forest, as well as all previous experiences of deer – including the notion that deer are a kind of animal, for instance – form the *external horizon* to which the deer is related.[2]

Again, the referential and structural aspects are merely analytical tools through which the researcher can structure their analysis and discussion of experience. The experiencer experiences the phenomenon simultaneously, as one.

The structure of awareness & categories of description

Each different way of experiencing a phenomenon can be characterized by considering the structure of the experiencer's awareness as they are aware of the phenomenon in that way. Each category in a phenomenographic outcome space describes a way of experiencing and can be analyzed in this way. By way of example, let us consider the first two categories in Eckerdal's outcome space concerning the concept

[2]Berglund (2005) observes that because the word 'horizon' in 'internal horizon' and 'external horizon' greatly emphasizes the border between the phenomenon and its surroundings, the terms are not always very intuitive. We agree, but persist with these terms here in the absence of existing better ones.

of object (which is shown in full in Table 7.1 on p. 97 above). In Category A, the structural aspect of the object phenomenon is experienced in terms of program code. The internal horizon of the structural aspect is the piece of code that defines an object, while the rest of the program code and its syntactical rules belong in the external horizon against which objects are discerned. The referential aspect is summarized in Table 7.1: an object is a piece of code. Other aspects of the programming endeavor (such as objects' runtime behavior) are marginalized and the experiencer is barely conscious of them. In Category B, by contrast, an object's role at runtime also falls within the internal horizon and gives new meaning to the phenomenon.[3]

The example above illustrates that a category of an outcome space can describe – and often does describe – a way of understanding in which the borders of the phenomenon are perceived differently than in the other categories. Does this mean that an outcome space actually describes not people's different experiences of a single phenomenon but multiple different phenomena? It might, if the categories were disjoint. Marton (2000, p. 105) reminds us that the phenomenon of interest to the researcher is constituted by the various ways in which it can be experienced. "These different ways are logically related to each other, it is in this sense they are experiences of the same object." For instance, in the example above, Category B is a richer variant of Category A; the critical features discerned in Category A are a subset of those discerned in Category B. It is up to the phenomenographer, in dialogue with his data, to delimit the phenomenon and conduct the analysis in such a way that the outcome space has a logical structure through which the categories relate to each other.

The structure of awareness & aspects of learning

Experience of each of the aspects of learning – the 'act', the 'why', and the 'what' – can be considered in terms of the structure of awareness (Figure 17.5). This permits us to use the structure of awareness to further break down our research question. As we ask "In what ways do programming students experience learning through visual program simulation?" we are asking several interrelated questions:

- What do programming students perceive as being the components involved in the act of learning through VPS and how are these components perceived as connecting? (the internal horizon of the act of learning)
- In what context do programming students perceive learning though VPS as taking place? (the external horizon of the act of learning)
- What does learning through VPS mean to programming students? (the referential aspect of the act of learning)
- In what situations do programming students expect to find a use for what they learn through VPS? (the internal horizon of the why of learning)
- In what broader contexts do such situations occur? (the external horizon of the why of learning)
- What is the eventual use of successful learning through VPS? (the referential aspect of the why of learning)

17.2 Phenomenography places loose restraints on research methods

Now that we have an idea of what we are going for, what does phenomenography say about concrete research methods? Nothing very prescriptive, although it does depend on who you ask.

17.2.1 Phenomenography is an 'approach' to research

> *Those who try to understand what a phenomenographer does risk falling for one of two misunderstandings: that phenomenography is a method which can be copied by following a sort of recipe, or that phenomenography is mystical and transcendent. Phenomenography is not a method which can be applied to a research question in some unproblematic way, nor does it involve introspection or meditation.* (Booth, 1992, p. 61)

[3] Eckerdal (2006) does not explicitly discuss her outcome space in terms of internal and external horizons; this is our interpretation of her categories.

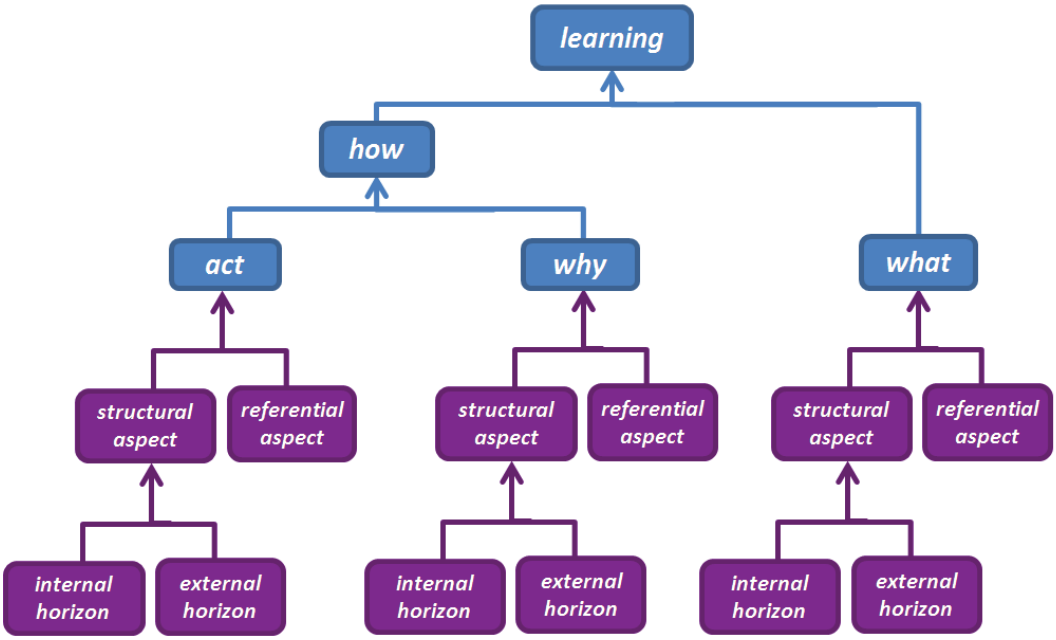


Figure 17.5: The structure of awareness applied to each of the aspects of learning. Adapted from Marton and Booth (1997).

Marton and Booth (1997) describe phenomenography as

> *a way of – an approach to – identifying, formulating, and tacking certain sorts of research questions, a specialization that is particularly aimed at questions of relevance to learning and understanding in an educational setting.* (p. 111)

Above all, they continue (p. 135), phenomenography is defined by its object of research, that is, human experience. Approaching research phenomenographically means seeking certain kinds of answers (primarily: categories of description) to certain kinds of research questions (primarily: how do people experience a phenomenon?). However, phenomenography, in the sense we use the term, does not impose methods for data collection or analysis.

That does not mean that phenomenography is entirely silent on these matters. "Since a research approach has its history, there is a tradition, or a network of competence, that the researcher partly can lean on in his selection to use one method [...] over another" (Berglund, 2005, p. 36). The phenomenographic literature does suggest that certain ways of going about data collection and analysis have served well. The phenomenographer can draw on this literature as he assesses how best to address his research questions in his particular context.

The following methodological advice is drawn in part from the phenomenographical literature and in part from the general literature on qualitative research methods.

Sources of data

> *Interviews are particularly well suited for studying people's understanding of the meanings in their lived world, describing their experiences and self-understanding, and clarifying and elaborating their own perspective on their lived world.* (Kvale, 1996, p. 105)

As the researcher does not have direct access to others' experience, he must content himself with other forms of empirical data. Any data that gives insights into people's experience can be used in a phenomenographic study. This data may include transcripts of interviews, video observations, others'

research findings, and, in principle, even physical artifacts and organizational structures (Marton and Booth, 1997). Although interviews are not the only option, they are, in practice, the one tried-and-true phenomenographic data source.

Phenomenographers like interviews because – as the word suggests – they enable an interactive exchange to take place about the views of the research subjects. In particular, they allow the researcher to follow up on interesting themes that emerge during an interview, and to ask the interviewee to elaborate on unclear points. This can be immensely helpful for gaining insights into the research subjects' experiential worlds.

Setting up interviews

Who to interview? Or might some existing interview data, collected for another purpose, do? Many phenomenographic studies – including ours – are directed towards improving pedagogical practice in a particular setting and other settings like it. In such "developmental phenomenography", Bowden (2000) argues, it is vital that the interviewees are learners in the particular setting. In this way it is possible to relate the partial understandings that emerge from the analysis to the educational setting of the students. Bowden also recommends that the entire study be guided from beginning to end by a phenomenographic mindset; interviews, for instance, should be conducted in such a way that they maintain a phenomenographic focus on experience. This precludes the use of interviews collected for another purpose. Some other phenomenographers are more open to combining phenomenographic analysis with non-phenomenographic data collection.

The selection of interviewees depends on the specific research goals, but also on the general goal of capturing the variation within the population of interest (Åkerlind, 2005b). With this in mind, a suitable strategy may be to choose a pool of interviewees who have different backgrounds or other characteristics (although they are from the same target population, which may have a shared background of some kind). This is consistent with the goals of *maximal variation sampling* (Patton, 2002, pp. 234–235).

How many people to interview? A downside to interviewing is that interviews, especially in-depth ones, are time-consuming to transcribe and analyze. However, practice has shown that for the purposes of a typical phenomenographic study, a fairly small number of interviewees is often sufficient. Ten to thirty interviews is a typical number. One way to proceed is not to decide the exact number of interviewees in advance, but to keep doing interviews until they start appearing similar and repetitive, and do not seem to reveal new perspectives. (Making a judgment on this requires doing data collection and early analysis in parallel; see below.) This does not guarantee exhaustiveness but is good enough for many practical purposes.

> *Inasmuch as a phenomenographic study derives its descriptions from a smallish number of people chosen from a particular population* [...] *the system of categories can never be claimed to form an exhaustive system. But the goal is that they should be complete in the sense that nothing in the collective experience as manifested in the population under investigation is left unspoken.* (Marton and Booth, 1997, p. 125)

Conducting an interview

The typical phenomenographic interview can also be characterized as a *semistructured* one: "It is neither an open conversation nor a highly structured questionnaire. It is conducted according to an interview guide that focuses on certain themes and that may include suggested questions" (Kvale, 1996, p. 27). Despite the thematic focus and prethought questions, "there is an openness to changes of sequence and forms of questions to follow up the answers given and the stories told by the subjects" (ibid., p. 124).

Marton and Booth (1997, pp. 130–131) urge the interviewer to guide the interviewee to a "state of meta-awareness" in which they reflect on their own understanding of the phenomenon of interest and articulate it as completely as possible. Marton and Booth observe that such an interview has elements of both social and therapeutic discourse. The latter aspect is more challenging and may require the interviewer to repeatedly bring the interviewee back to reflect on certain matters, or to offer interpretations of the interviewees' statements for the interviewee to examine and possibly reject. Marton and Booth

further note the importance of establishing a comfortable relationship between interviewer and interviewee in a phenomenographic interview, as affective issues may need to be addressed and defenses overcome for the interview to reach an open, self-reflective state (see also Bowden and Green, 2005).

> *The questions asked are of decisive importance, of course. Surely, it takes bright ideas and creative insights to formulate questions that stimulate openness.* [...] *The fact that we are interested in* what *the students think about and* how *they think, not in whether they manage to produce the right answer should be obvious to them as well. The interview should be an open-minded exploration of the landscape of thoughts, not an examination nor an instructional session.* (Johansson et al., 1985, p. 252)

Kvale (1996, pp. 133–134) enumerates types of questions for qualitative interviews. *Introducing questions* open up the interview by introducing a broad topic of conversation, *Follow-up questions* seek to get the interviewee to expand on a word or topic that has come up. *Probing questions* request elaboration. *Direct questions* request information about a particular aspect that the interviewee has previously not spoken about. They are usually best left to the latter part of the interview. *Indirect questions* request the interviewee to reflect on the topic from a different perspective, such as that of other students; the answers can be hard to interpret. *Structuring questions* move the conversation in a new direction when a theme has been exhausted or because of time pressure. Simple *silence* often works in getting the interviewee to elaborate on something or to open up a new line of dialogue. Finally, *interpreting questions* present an interpretation of what the interviewee has said back to them for them to comment on.

Any or all of these types of questions may be employed in a phenomenographically motivated interview. For interviewees to articulate their experiences as completely as possible, follow-up and probing questions are especially important. They can be used to encourage the interviewee to explain their understandings of aspects of the phenomenon that they have mentioned, and to elaborate on how they see particular aspects or their relationships. Åkerlind (2005a,b) emphasizes the need for the phenomenographer to get beyond 'what' questions to ask 'why' questions and get at the reasons behind how the interviewees think. Bowden (2000) encourages the use of interpreting questions in a phenomenographic interview. In particular, he suggests that pointing out apparent inconsistencies in the interviewee's ideas can be a fruitful way of getting the interviewee to speak about their understanding of the phenomenon.

Marton (1986, p. 42) prefers "questions that are as open-ended as possible in order to let the subjects choose the dimensions of the question they want to answer." On a related note, Bowden (2000, 2005) suggests that questions of the form "what is X?" may work against the goals of phenomenography, and that it is not always necessary for the interviewer to explicitly mention X (the phenomenon) at all when putting questions to the interviewee. Instead, Bowden promotes the use of problems related to the phenomenon under study during the interview – for instance, when seeking to find out students' understandings of a physics concept, students might be invited to solve a physics problem and discuss it with the interviewer. This gives the interviewee the opportunity to choose the aspects of the phenomenon that are most relevant (which the interviewer may then pick up on), and helps to avoid the problem of the interviewer putting words into the interviewee's mouth. The concrete context of the problem further helps the interviewer and the interviewee to establish agreement about what is being discussed.

Overlapping data collection and analysis

In qualitative research in general (see, e.g., Patton, 2002, pp. 436–437), and phenomenography more specifically (see, e.g., Marton and Booth, 1997, p. 129), analysis may begin while data collection is still ongoing. This brings the risk of jumping to premature conclusions and inadvertently affecting the later interviews. Because of this, some phenomenographers (e.g., Bowden, 2005) object to starting analysis before all the interviews are complete. However, doing data collection and analysis in parallel has the great advantage that early phases of analysis can inform later parts of data collection. For instance, later interviews may be adjusted to explore in greater depth some themes that early analysis suggests are important. Such flexibility allows the researcher to take advantage of important insights that occur during the data collection phase.

Analysis: individuals ⟷ themes

Marton outlines, in general terms, a process that may be used for constructing outcome spaces (Marton, 1986; Marton and Booth, 1997). We have paraphrased this advice below in an itemized format, which is not to be interpreted strictly algorithmically; analysis may flow back and forth in a natural way between the 'steps' listed below.

1. Interpret each utterance by assessing its meaning against the individual context (e.g., interview transcript) that it is a part of.
2. Select which utterances appear to be relevant for answering the research question.
3. Abandon the boundaries separating individuals (for now). Shift your attention to the meaning in the selected quotes. A single, large 'pool of meaning' forms.[4]
4. Look for themes in the pool of meaning.
5. Interpret each quote against this pool of meaning, without forgetting the individual context that the quote comes from.
    - Do not judge the quotes in relation to what the phenomenon 'is really like' or your own experience of it. Instead, assess the quotes in relation to how they fit in with how the utterer appears to experience the phenomenon. Be open to others experiencing it in different, surprising ways.
    - Vary between a focus on individuals and a focus on themes. Alternate between the thematic and individual contexts as you examine a quote.
    - Look for consistencies within what an individual says. Assume that what people say is logical, given a particular way of understanding the phenomenon.
6. Sort quotes (rather than individuals or full transcripts) into categories based on their similarities to and contrasts with other quotes.
7. Make explicit the criteria for the provisional categorization.
8. Check the categories against the entire data.
9. Refine the categorization iteratively. Finish when the categorization is well defined and does not change between iterations.

Some qualitative analyses start with the construction of case studies of individuals. Other qualitative analyses start with a thematic analysis across individuals. The form of analysis suggested by Marton interleaves the two, requiring the researcher to move fluidly between individual and thematic contexts. Patton (2002, p. 440) warns about the difficulty of qualitative analysis that attempts to do work on cases and themes at the same time. Within the phenomenographic literature, Bowden (2000) is concerned with the same issue. He prefers that entire transcripts (e.g., interviews with a single individual) rather than quotes are sorted into categories. Bowden argues that using entire transcripts makes it easier to be mindful of the individual context in which each utterance must be interpreted.

Åkerlind (2005c) further discusses variants of the phenomenographic analysis process. In addition to using different units of categorization, phenomenographers vary in how much importance they give to the goal of forming logically related categories, and in how they see the role of researcher collaboration in phenomenographic analysis (see also Bowden and Green, 2005).

[4] The phenomenographer ultimately looks for the meaning behind particular utterances, rather than the ways in which those utterances are worded. Language is nevertheless often a supremely important aspect of phenomenographic analysis, as the phenomenographer has to consider how the research participants use language and what words mean to them (Bowden and Green, 2005, pp. 86–87, and references therein). In a field riddled with technical terms (such as programming) it is perhaps especially important to remember that people, and novice learners in particular, may use words in a sense quite unlike what the content expert might expect.

Whichever variant of the analysis process is adopted, analysis involves consideration of both individual and collective perspectives. What about the results? Many – probably most – phenomenographers maintain that the results must be interpreted on a collective level. An individual's understanding changes with time and context (see, e.g., Marton and Pong, 2005), and it is not meaningful to assign individual research participants to the categories.

> *The individual student comes to serve as a "carrier" of (fractions of) one or many ways of understanding something. The outcome of the project then describes the ways in which something is understood within a cohort. The individual voices have "disappeared" from the final result. What remains is the researcher's interpretation of what they have said. Thus, the result can only be interpreted for a collective.* (Berglund, 2005, p. 38)

The target: a good outcome space

What is a good outcome space like? Marton and Booth (1997, p. 123) warn us that "There is no complete, final description of anything and our descriptions are always driven by our aims". Instead of completeness and finality, Marton and Booth (ibid., pp. 125–126) present three criteria for judging the quality of an outcome space, which we interpret as follows.

1. The outcome space must clearly relate to the phenomenon of interest in a way that meets the goals of phenomenography. That is, each category should be a distinct way of experiencing the phenomenon. The categories, taken together, should describe variation in experience rather than be some other kind of categorization of how people think or behave.
2. The categories have to be logically connected to one another, usually in a hierarchical structure.
3. The outcome space should be parsimonious. Only the critical aspects that are needed for describing the variation between ways of understanding or for delimiting the phenomenon should be described.

The second criterion is an often-debated aspect of phenomenography and worth elaborating on.

Analysis: seeking logical structures

> *Educationally, it is a reasonable assumption that there is a norm, a particular way of experiencing a phenomenon that is to be preferred over others, and that is what the educational effort is designed to foster.* [...] *Thus, we seek an identifiably hierarchical structure of increasing complexity, inclusivity, or specificity in the categories, according to which the quality of each can be weighed against that of the others.* (Marton and Booth, 1997, p. 126)

The researcher chooses and delimits the phenomenon of interest. He has an initial notion of what he wishes to investigate, which is affected by his own understanding of the phenomenon. The limits of the phenomenon are molded during analysis as the researcher's dialogue with the data highlights aspects that the subjects associate with the phenomenon.

Each of the resulting categories of description should highlight some facets of the phenomenon of interest. Together, the ways of experiencing captured in the categories constitute the phenomenon; in terms of the outcome space, the phenomenon is 'that which can be experienced in these different ways (and possibly others that did not emerge in this study)'.

To ensure that the results do indeed describe a single phenomenon (and are more readily intelligible and useful), the researcher should try to identify categories that are logically connected to each other. In practice, "logically connected" effectively means that each category is a grouping of aspects of the phenomenon and relationships between aspects, and that each such grouping is a subset of some or all of the other categories. A single top category usually groups together all the critical aspects and relationships identified. It is the researcher's responsibility to delimit the phenomenon and conduct the analysis in a way that ensures that such logic emerges. "Inevitably, there is tension between being true to the data and at the same time creating, as the researcher sees it, a tidy construction which is useful for some further explanatory or educational purpose" (Walsh, 2000, p. 21).

Here it is important to stress the intentionality of the researcher and how it relates to the resulting categorization. Categories of description (that is, ways of understanding) are an analytical product constructed by the researcher, who relies on a particular research framework, has delimited the phenomenon in a particular way, and examines it from a particular perspective affected by his overall research goals, and on a particular level of abstraction. The categories do not pre-exist analysis, waiting for the researcher to pick them out from the data. Instead, they emerge from the relationship between the data and the researcher (Walsh, 2000).

Such analysis runs several risks, including "adding or adjusting categories where this is not supported by the data; imposing a logical framework on the data where this is not justified; and analyzing the data from the researcher's or content expert's framework, so that the interpretation of the data is skewed towards an accepted or expert view of the phenomenon" (Walsh, 2000, p. 23). The main way to address these challenges, Walsh argues, is alertness to them on the part of the researchers.

Phenomenographers differ in how much emphasis they place on the importance of logical relationships (Åkerlind, 2005c; Bowden and Green, 2005). Often, the emergence of a logical structure is seen as a criterion for a good outcome space. The lack of a such structure to the categories raises questions about the quality of the analysis and about whether the results do indeed describe ways of understanding in the phenomenographical sense. The lack of structure is a warning to the phenomenographer, and needs to be addressed when reporting results, unless a logical structure emerges through further reflection on the data (Berglund, 2005, p. 84).

Establishing reliability

Multiple positions on how to establish reliability (or "precision"; see Section 16.3) have been advanced within the research community (see, e.g., Åkerlind, 2005c; Bowden and Green, 2005; Sandberg, 1997). An established practice is to use a form of interjudge reliability check in which other researchers are given the data and an outcome space that has been produced by a lead researcher who has analyzed that data. The other researchers check whether they can also 'see' the same categories within the data as the lead analyst. Bowden (e.g., 2005) promotes a team process in which multiple analysts discuss and mutually critique each other's interpretive hypotheses. Sandberg (1997) is critical of all forms of interjudge checks in phenomenography on the grounds that such checks do not help to critique the quality of the data collected and that they are a poor epistemological fit with phenomenography. Sandberg prefers instead that the researcher concentrate on making their personal interpretive process as explicit as possible.

17.2.2 So it is not a method?

It is not rare to hear phenomenography referred to as a method or methodology. Marton, the father of phenomenography – who has more recently stated that phenomenography should be primarily defined by its research goals and not by its use of methods (see p. 269 above) – has also previously referred to phenomenography as a research method.[5]

The reason for this ambiguity may be historical. Above, I have introduced present-day phenomenography as a research approach that answers a particular kind of question with a particular kind of answer, and is founded on theories of awareness and variation. However, the first studies which are now termed phenomenographic predate the term itself and the theories on which current phenomenography is based. Phenomenography originates not from theory but from empirical studies in the 1970s of students' approaches to learning (e.g., Marton and Säljö, 1976), which gave rise to the notions of deep and surface approaches to learning. (These early findings remain the most broadly influential phenomenographic results.) Only afterwards were the theoretical frameworks of variation and awareness introduced by Marton and his colleagues. One might say that phenomenography has evolved from a research method used in the 1970s into a theoretically laden, methodologically varied approach to empirical research on human experience. Historical perspectives on phenomenography have been provided by Pang (2003) and Richardson (1999).

[5]The elusive definition of phenomenography has caused some gray hairs to emerge and gray cells to suffer, not only in our own research group but among PhD students elsewhere, too (Ireland et al., 2009).

Different phenomenographers define phenomenography differently, and there are distinct movements within the phenomenographic tradition, some of which are more closely tied to specific research methods than others are. Some of these variants could well be called methods or methodologies. Hasselgren et al. (n.d.) give an overview of the main flavors of phenomenography (see also Åkerlind, 2005c; Bowden and Green, 2005).

Enough generic talk.

17.3 Here is what we did in practice

17.3.1 We collected data from interviews with programming students

The way we conducted the interviews accommodates many of the considerations from the previous section.

The interviewers

To improve the trustworthiness of our research, we wished to have an interviewer who was independent of UUhistle's development. On the other hand, the author of the thesis would be doing much of the analysis, some of it during the interviewing process, so we also wished to involve him in the interviewing so that he could make use of the early analysis to explore emergent themes in more detail.

Consequently, two researchers collaborated to do the interviews. One interviewer, Jan Lönnberg, interviewed five students, and the author of the thesis interviewed six, making eleven interviews in total. The author was in charge of student selection.

Both interviewers had some experience with phenomenographically motivated interview studies from earlier projects.

Student selection

Our interviewees were students taking the spring 2010 offering of CS1–Imp–Pyth described in Section 16.4.

A couple of weeks into the course, all the students of CS1–Imp–Pyth answered an online questionnaire in which they were asked to rate their programming background, their attitudes towards programming, and the workload and difficulty of the CS1 course so far. We used the answers to this questionnaire, and the results from the assignments submitted, to mold our interviewee selection process so that we got a mix of interviewees with different backgrounds, attitudes, and estimates of course difficulty, as well as a mix of scores from the course assignments. Such a selection, we felt, would give us a good basis for exploring the variation in ways of experiencing. The interviewees also came from a variety of degree programs within engineering.

We surmised that students who had a harder time with the course or were less enthusiastic about it would have more trouble seeing VPS in a rich way and might be a good source of data for studying partial understandings of the phenomenon. We therefore skewed our selection process towards such students, inviting more of them and fewer of the experienced and motivated students. The selection process was unformulaic and involved randomly picking out names from a list until we had what appeared to be a suitable mix.

Given our research interest, we ensured that all our invitees had submitted an answer to at least one VPS exercise; most had submitted many.

The interviews were conducted partially in parallel to the experimental study described in Chapter 19. We checked that none of our invitees had been part of the experimental group in that other study.

Both the interviewers had taught at our university in past semesters. We confirmed from our department's records that none of the students was interviewed by a researcher that the student had previously taken a class with.

We aimed for a minimum of 10 interviews. From earlier experiences of interviewing CS1–Imp–Pyth students for another project, we knew that far from all of the invitees would agree to being interviewed. The total number of invitations thus significantly exceeded that of the interviews we needed.

Invitations

Again wishing to introduce variation, we sought data from different phases of the course, with some interviews being conducted nearer the beginning of the course and some at the end. We decided to invite students in stages, adapting to the situation as we went along, depending on the acceptance rate of the invitations and the early stages of analysis.

We sent the invitees a fairly informal email that invited them to participate. Students who did not answer the invitation were sent another email a few days later; those who did not respond to this one either were not approached again.

The invitations stressed that the interviews were not a part of CS1–Imp–Pyth and that they would be handled confidentially. It was explained that the interviews were about the course assignments, but VPS was not singled out as a topic.

The participating students received two movie vouchers as remuneration for a half-hour interview.

In total, we invited 31 students, of whom 11 accepted the invitation: 4 students around the fourth week of the course, 5 around the sixth week, and 2 around the eleventh week. In our estimation, the set of eleven interviewees satisfied our goal of 'variety with an emphasis on strugglers'. Six of the interviewees were female, five male. Most were around 20 years of age; one was an older student. Only one of the students had prior programming experience beyond the quick brush with Visual Basic that some of the other interviewees had had.

Interview (semi-)structure

The interviews were semi-structured (see Section 17.2 above), and involved a great deal of improvisation. The following gives a general feel of how the interviews progressed. Specific interviews differed in their details.

Each interview started with introductions of the participants. We made a conscious effort to keep the atmosphere casual and conversational, and the interviewee relaxed. One interviewer invited the interviewees to his office, the other interviewer arranged the interviews on the 'neutral ground' of a small meeting room. During many interviews, we offered a selection of soft drinks and water for the interviewee.

We asked the students for permission to record the interview. The interviewer briefly explained that our research aims to improve the teaching and learning of programming, and reiterated that the interview would be treated confidentially and anonymously, and would not affect their course grade.

The interviewer then directed the conversation towards the course assignments and towards the VPS assignments in particular. Once the topic of VPS was mentioned, the interviewer suggested that it would be easier to discuss it through a concrete example. He then asked the interviewee if they would mind showing and discussing how they work on a VPS exercise on a computer provided by the interviewer.

It was suggested to the student that they work on the next VPS exercise from CS1–Imp–Pyth that they had not previously done. Failing that (because the student had already done all the assignments they had the prerequisite background for), we suggested one of a selection of small VPS exercises prepared for the purpose of the interviews.

The interviewee mostly worked on the exercise without help from the interviewer. We only gave small hints for solving the exercise when the learner got completely stuck, and only after discussing what the problem was. On occasion, if an exercise seemed to be too difficult, we switched to a simpler example program before returning to the original exercise.

Much of each interview focused around the VPS task. The student would work on the exercise, thinking aloud. Frequently, we would ask questions about a pertinent topic. A discussion, sometimes lengthy, sometimes short, would ensue before the student returned to work on the exercise. These discussions would typically start from questions directly related to the VPS exercise, such as "What are you doing now?", "How do you think about what you just did there?", "What do you think about as you

choose what to do?", "What does this [visual element] here mean to you?", and "What happened there? What does that mean to you?".

Many of our questions were prompts for more information on something that the student had mentioned before: "Can you say more about that?", "What do you mean by X?", etc.

We attempted to get the interviewee to talk as much as possible, while talking as little as possible ourselves. We avoided introducing new concepts or vocabulary into the interview but waited for the interviewee to do so. In several interviews, this was quite challenging, however, and we had to compromise and use more direct questioning on occasion. For instance, we sometimes had to introduce a learning perspective into the conversation with a question such as "Have you learned something from this?" to get the interviewee to talk about matters of interest. (Ideally, we would have preferred any mention of learning to originate from the interviewee.)

Another pattern to our question-setting was an attempt to approach the same topic from multiple angles by changing the context. In particular, we approached the issue of learning in the concrete context ("Did you learn something from what you've done right here?"), other contexts in past experience ("Can you think of some other UUhistle exercise where you learned something?"), a more general course context ("Would you say it's useful for someone somehow that you have this kind of assignment in the course?", "Does this kind of assignment seem meaningful to you?"), and the context of an imagined conversation with a friend ("How would you describe the point of these assignments to a friend, if they asked you?"). By using multiple contexts in this way, we sought to get a richer articulation of the interviewee's experience.[6]

We sometimes, in an improvised manner, used interpreting questions to get the interviewee to confirm or disconfirm our initial interpretations, or to get them to elaborate on their perspective. This included pointing out apparent inconsistencies in what the interviewee had said (e.g., the interviewee might alternate between saying that the VPS exercises are useful and that they are not).

Near the end of the interviews, we asked the students directly to comment on the relationship between the VPS exercises and program-writing activities, unless they themselves had spontaneously brought up that topic before.

At the end of the interview, the interviewee could comment freely on anything they wished to bring up and could ask questions. The interviewer also occasionally took the initiative to teach about some programming topic that the student had had trouble with during the interview.

All the interviewees agreed to being recorded. We recorded the sound and the onscreen activity from each interview. The recording quality was good, and we were able to include all the recordings in our analysis.[7]

On interview quality

We note in hindsight that we ended up spending rather too much of most interviews on making progress through the entire VPS exercise and prompting the interviewees ("What next?"), a pattern that the students also easily fell into. We were perhaps not quite focused enough on the research question that the interviews were primarily designed to answer (stated above); a lot of the material from the interviews involves details about specific simulation steps and does not contribute a lot towards answering the main research question. If we were to do the interviews again – or to extend this work later – we would attempt to keep the interviews more focused on a still deeper discussion of how students perceive VPS and UUhistle in general. Nevertheless, we obtained data that we feel is quite satisfactory for phenomenographic analysis.

[6] This question-setting strategy, which we found very helpful, was recommended to us by Anders Berglund (personal communication).

[7] It is perhaps worthwhile to mention that one of the interviewees was somewhat drunk at the time of the interview. However, in the interviewer's judgment, the inebriation – although visually and olfactorily appreciable – did not significantly affect the conversation, and the interview remained focused throughout. In the absence of authoritative methodological advice regarding bibulosity in a qualitative interview setting, the data from this interview was included in the analysis as per usual. We have found no reason to believe any of the categories in our outcome space emerged as a result of intoxication. Even though the interviewees have been anonymized, it is perhaps best to err on the side of caution and not to identify which of the interviews is the one in question.

Supplementary observational data

In addition to the interviews, we analyzed recorded observations of student pairs working on VPS exercises. This data – also in the form of sound-and-screen captures – had been collected primarily for the studies described in Chapters 18 and 19; the data collection process is also described there.

We wanted to use the video observations as a secondary data source to enlarge our data base and to enrich it with material from a setting that is more natural than an interview.

17.3.2 We analyzed the data to come up with logically connected categories

Nearly immediately after the first interview finished, our analysis of the data started (as an activity in its own right; in a sense, data analysis starts even during the interviews). The early stages of analysis interleaved with the data collection.

Researcher roles

Three researchers participated in the analysis. The lead researcher – the author of the thesis – had the primary responsibility for going through the data and coming up with draft categorizations. He was also in charge of transcription and reporting. The two other researchers – Jan Lönnberg and Lauri Malmi – had the role of critical discussants: several times during the analysis process they commented on the drafts and suggested clarifications, modifications, and improvements for the lead researcher to consider as he refined the analysis.

Dealing with the recordings

During the first pass through the data, the lead researcher watched each recorded interview, made notes about the topics of conversation covered, and paraphrased the discussion. On the basis of these notes, he chose the sections of potential interest for answering our research question, and transcribed those sections verbatim. These selected sections of interest in the interviews served as the primary data for the analysis.

This approach saved the trouble (or expense) of transcribing the entire material – which contained a lot of repetitive detail about specific VPS steps and other discussion about topics that were uninteresting from the point of view of our research. However, it came with the risk of missing out on interesting data that did not seem important at the beginning of the analysis. To ensure that this did not happen, the lead researcher watched the recordings again at a late stage of the analysis, looking for any previously ignored sections of interest. (None were found.)

The observational data of student pairs working on VPS served a supplementary role. The lead researcher watched the recordings once, noting sections of potential interest, reflecting on how they related to a draft of the outcome space, and considering whether the observations suggested any new categories beyond what we had outlined on the basis of the interviews. The observational recordings were not transcribed, apart from the sections quoted below as we present our results (in the next two chapters as well as this one). As things turned out, the impact of the observational data on this chapter's study was minor and resulted in no new categories.[8]

Categorizing quotes

The way we analyzed the interviews was based on the advice given in the literature, outlined in the previous section. We took a general outline for the analysis process from Marton's advice (p. 273 above). This choice means that we treated individual quotes (rather than whole transcripts) as the unit of categorization. Nevertheless, we sought to be mindful of the individual context from which each quote was taken. While keeping track of the thematic and individual contexts at the same time is challenging, we felt that it was feasible, given the relatively small amount of data we had to handle.

[8] We did not find the observations to be a very good source of data for answering our phenomenographic research question, since, in the absence of an interviewer to probe deeper, the discussions between students were rarely of the sort in which ways of experiencing were articulated. The observations were much more useful for answering some other questions – see the next two chapters.

We sought to form an outcome space whose categories are logically connected in the sense described in Section 17.2. Constructing the categories and crystallizing their precise definitions and relationships took many iterations. The categories in the final version of our outcome space form a hierarchical structure, which, in our estimation, is logical and at the same time does justice to the data.

Sensitizing concepts

During the analysis, we made use of the theoretical constructs from Section 17.1: the aspects of learning and the structure of awareness. These constructs were brought into the analysis process from outside the data and served as a *sensitizing framework* (Patton, 2002) in our study. We used them to inspire our analysis process, to suggest places to look for interesting aspects, and to give structure to our characterization of each of our categories and their relationships.

Some other phenomenographical studies have focused on a single one of the 'act', 'why', or 'what' aspects of learning and reported outcome spaces pertaining to just that aspect. Others still have studied both the 'act' and 'why' separately and reported distinct outcome spaces for each one. As our interest was on the 'how' aspect of learning – which comprises the 'act' and the 'why' – we, too, considered the latter option. However, during the analysis we came to feel that in our data the 'act' and 'why' of learning are very tightly integrated – motivations for doing VPS relate directly to the ways in which the act of VPSing was perceived – and we wished our results to reflect this. We therefore formed a single outcome space that encompasses the entire 'how' aspect of learning. Within each category, nevertheless, we have analytically separated the 'act' and 'why' aspects and the structure of awareness that pertains to each of them. This approach allows us to structure our findings through the sensitizing concepts, while at the same time emphasizing that the 'how' aspect of learning is perceived as an integrated whole.

This approach is reflected in how we present our results, next.

17.4 Students experience learning through VPS in six qualitatively different ways

From our analysis of the data, an outcome space of six logically connected, qualitatively different categories of description emerged. We have labeled these with letters from A to F. Category A is the least inclusive and describes the simplest way of understanding what it means to learn through visual program simulation. Category F is the most inclusive and describes the richest way of understanding we found. The other categories fall in between.

An overview of our outcome space is given in Table 17.1. Figure 17.6 illustrates the relationships between the categories. The following subsections elaborate on each category in turn. While reading, the reader may also wish to peek ahead at Table 17.2 on page 292, which characterizes each category in terms of the aspects of learning and the structure of awareness.

On names and genders

To preserve student anonymity, we have randomly assigned each of the students quoted below one of the first names – Beth, Elizabeth, Jan Erik, William, Robert, Lynda, Sue, Otto, Raymond, Morten, Kate, and John – of the authors of an influential CER paper. Of these, we have used the names Kate and John with subscripts to refer to the paired students whose work we observed on video; for instance, Kate_{17} and John_{17} are the two students that form pair number 17. (Pair numbers are essentially random.) The other names we have used for the students who participated in the in-depth interviews described above.

We considered gender irrelevant during the analysis. We are not aware of any gender-related issues in our results. The random names we have given to students frequently do not match their actual gender.[9] Our use of personal pronouns below is in accordance with the aliases we have given.

Interviewer_1 is Jan Lönnberg. Interviewer_2 is Juha Sorva.

[9] Readers who are fans of Cash (1969) may be interested in knowing that the student known as "Sue" is actually male.

Table 17.1: Ways of perceiving what it means to learn through VPS, in brief. For more detail, see Figure 17.6 and Table 17.2.

Category	Learning through visual program simulation is perceived as...	And as in Category
A	...learning to manipulate graphics in order to complete course assignments.	
B	...learning about what the computer does as it executes programs.	A
C	...gaining a new perspective on the execution sequence of programs within the computer.	B
D	...learning to recall programming commands and patterns of code.	A
E	...learning to understand programming concepts present in example code.	D
F	...learning programming via learning about program execution.	C E

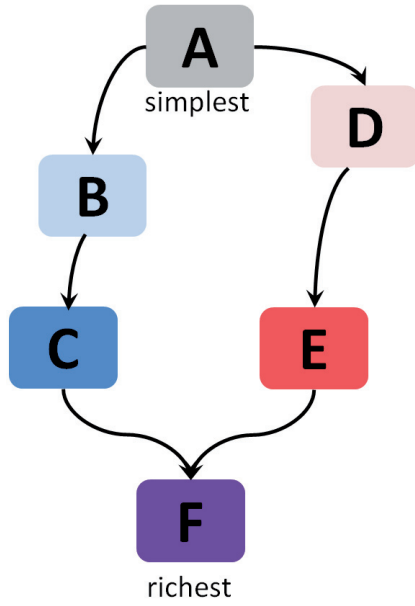


Figure 17.6: Logical relationships between the six categories. Each arrow indicates a relationship where a richer way of understanding is an extension of a simpler one. For example, Category F extends both C and E (and, transitively, all the other categories as well).

17.4.1 A: VPS is perceived as learning to manipulate graphics

Category A describes a way of experiencing learning through visual program simulation as *learning to manipulate graphics in order to complete course assignments*.

In Category A, awareness is focused on the concrete. The act of learning through VPS is experienced primarily in terms of what one sees (graphical elements, program code) and does (e.g., moving elements around). A vague relationship between visual elements and sections of program code is discerned.

The topic of frames has come up in Robert's interview:

> **Interviewer$_1$:** What do these frames tell you?
> **Robert:** *Pauses for a bit.* I don't know.
> **Interviewer$_1$:** Uh-huh. Okay.
> *silence*
> **Interviewer$_1$:** You can't say more than... you clicked on, yes, what is that?
> **Robert:** Well, these *(points at frames)* are things at different levels... That you move around here *(points at the stack)*. Like, you don't dump them all into the same square... I mean, box.
> **Interviewer$_1$:** Yeah. What do these boxes mean, then?
> **Robert:** *Points at a function definition in program code.* They are like some of that kind of stuff, at one level. Maybe.

Robert is focused on what you *do* with each element.

> **Interviewer$_1$:** In general, what kinds of things have you seen there in the heap?
> **Robert:** Well, what gets offered there.
> **Interviewer$_1$:** Mmm. Offered... wha...?
> **Robert:** *Interrupts.* Well, like when these things *(points at the entire code)* get asked, then these *(points at the heap)* are what you answer.
> **Interviewer$_1$:** Okay. So what do they mean for the execution of the program?
> **Robert:** Nothing. It's just to... I don't know.

In this view, the act of learning in UUhistle involves the program code, the visualization and its components, the operations that the student can perform on the visualization, and the student himself. The visualization is seen as an aspect of the VPS system. It provides affordances for the student to perform actions as required – to "answer the code", as Robert puts it. The code, the visual elements, and the operations are further perceived as being related to rules or requirements set by the VPS system, which determine what the correct operation is at each stage of the exercise. VPS is seen as a sort of 'game of boxes' in which the student aims to manipulate the boxes to get forward in the game and eventually to complete it.

Robert appears preoccupied with "getting to jump forward" in the assignment, something that he struggles greatly with. He observes that one eventually learns what do to deal with the system.

> **Robert:** Of course when you know what you must do, then it gets easier, but for me it's difficult to get to jump forward in these things.

Another student also feels he has learned something about what to do.

> *Raymond is working through some simulation steps very fluently after some earlier trouble.*
> **Interviewer$_2$:** It seems to feel pretty consistent to you, how it goes?
> **Raymond:** Well, yeah, once you do it wrong once then you maybe remember the next time.

The quotes from Robert and Raymond illustrate how learning in a VPS context is perceived as directed towards the VPS activity itself. VPS work is perceived as increasing one's ability to follow the system's rules and perform the correct graphical manipulations. Doing the assignments teaches about doing the assignments. In Category A, this is the only reason for learning within VPS. Robert comments on UUhistle and its role in CS1–Imp–Pyth with distaste:

Robert: Yeah, well, this UUhistle is completely awful, but this course is basically pretty well run. [...] This UUhistle is, like, the biggest flaw in the whole thing, because it's no use at all.
Interviewer$_1$: Uh-huh.
Robert: It only does harm because it's annoying.

John$_{37}$ voices a similar sentiment as he and his partner begin to work on a VPS exercise:

John$_{37}$ clicks around seemingly at random in the UUhistle GUI.
John$_{37}$: What! ...okay...first...What the fu...Why do we have to do these incomprehensible things?
Continues to poke around in the GUI.
John$_{37}$: *In dismay:* Nonononononono. You know, I've done all of these so that I just fucking tried every option.
Kate$_{37}$: Mmm.
...
John$_{37}$: *With deliberation, straight into the microphone:* These assignments appear slightly silly.

According to Robert and John$_{37}$'s comments, visual program simulation exercises are pointless. Similarly, what satisfaction Robert gets from doing a VPS derives from being finished with the task:

Interviewer$_1$: So, does this seem meaningful, this kind of exercise?
Robert: No. *Pauses.* Except if you get it done in one go, by accident.
Interviewer$_1$: Uh-huh. Okay. You don't feel that you learn anything from having to do these?
Robert: No, because you get so annoyed that you just start doing it by force and once you get rid of it, you're pleased.

Any link between simulation steps and programming is theoretical and vague at best. In the following excerpt, Raymond and the interviewer are discussing whether Raymond reflects on what he has done after finding the right simulation step:

Interviewer$_2$: So you just move forward?
Raymond: Yeah. You don't think about it much. Like, I don't know about this mouse-clicking, if it has anything to do with programming.
Interviewer$_2$: Yeah.
Raymond: *Perhaps misunderstands the interviewer's "yeah", which was intended to be non-committal, as affirmative.* Yeah? Like, I have to click on this "false"[10], and...what the connection to programming is? Like, I can't come up with anything that corresponds to it.

Near the end of the interview, the interviewer returns to the topic with a more direct question:

Interviewer$_2$: So, does this affect how you think when you program, like, when you do the [program-writing assignments]? Does this reflect on that?
Raymond: No...not really. *Pause.*
Interviewer$_2$: So this is, so to speak, a pretty separate thing?
Raymond: Yeah. Yeah. This is what you get the points for, when you just do it.

In this view, the need to learn arises out of the need to complete VPS assignments and score points. Or, to rephrase in phenomenographical terms, the assignments one must do comprise the internal horizon of the why of learning, which is seen against an external horizon of the educational setting (see Figure 17.2).

[10] UUhistle does not actually ever require the user to click on a boolean value.

17.4.2 B: VPS is perceived as learning about what the computer does

> **Interviewer$_2$:** How about if your friend, who's never seen this UUhistle thingy, asked you what this is, what it is that you see here?
> **Lynda:** I'd say that this is the computer. What the computer does when... when I click to make it run the code. This is what the computer does by itself.

In Category B, the visualization used in VPS is perceived as being related to programs' execution within a computer. Learning through VPS is seen generally as *learning about what the computer does as it executes programs*.

> *Pair 20 have just started a VPS exercise. John$_{20}$ explains what is going on to his partner, who has never done an assignment in UUhistle before.*
> **John$_{20}$:** These are, like, you basically... you, like, try to think of what the computer... like, what the computer would do if those comman-... instructions had been given, so what would happen within, how it, like, the computer, executes the program.
> **Kate$_{20}$:** Right. Okay.
> **John$_{20}$:** Here's, like, what happens in there. *Waves the mouse cursor about the call stack.* And here's what it, like, prints out. *Motions at the output console.* And... we have to, like, do it by dragging things ourselves... *Motions at the operators and the heap.* Let's see how we get started.
> *They soon return to the topic as John$_{22}$ is manipulating a frame to simulate the line* `first = [10, 5, 0]`*:*
> **Kate$_{20}$:** So here we're supposed to create what it looks like, what it prints out?
> **John$_{20}$:** Yeah, well, that goes there. *Motions at the output console.* Here we're supposed to do what, like... how the computer, like, handles it. Like, we have 'first' there, so we have to create a new variable.

In this view, the visual elements are seen as relating to the computer's view of program execution. Lynda comments on the elements representing functions, for instance:

> **Interviewer$_2$:** Now the code is ready, here. And then there's these pink boxes there...
> **Lynda:** These here *(points at function boxes)* are... how the computer sees them, as boxes like that.

Lynda, here, sees the visualization in terms of how the computer carries out the program. We may also conjecture that this view is present in some of the thoughts that occur to her as she works on the VPS exercise. For instance, at one point, she becomes concerned with the question of how the computer (or UUhistle, which presents the computer) keeps track of a parameter value, and asks "How does it know what `name` is?" (The question is resolved as she realizes she herself has just passed the appropriate value of the `name` parameter into an upper frame.)

Whether it is useful to know what the computer does is a different matter. Even though she muses vaguely that she should know something about the computer, Lynda does not find the VPS assignments meaningful.

> **Interviewer$_2$:** How would you compare the UUhistle assignments and [the program-viewing assignments]... their relevance to the course, or what you learn from them, or...?
> **Lynda:** Well, somehow the [program-writing assignments] seem quite a bit more meaningful, because that's the real thing, what you need to do yourself. Because this here is only about what the computer does... and at least I'm not terribly interested in that. In what the computer is actually up to. For me, it's much more important to be able to create a program myself... Even though I should know what the computer does, too.

Jan Erik also sees UUhistle as a system that makes the computer internals explicit. He interprets the visualization in terms of computer memory.

> **Jan Erik:** They [the boxes in UUhistle's view] are some kind of wholes that are connected to each other somehow, here in memory. I genuinely don't know...
> ...
> **Jan Erik:** *Laughs.* I have a feeling that...I don't know any better, so I'd imagine this is pretty much like how they are there...stored there. In this way. Well, not exactly as boxes but...I can't really say...Well, there are these structures there, floating about in memory, I suppose. We have these variables, which have some values, and they've been put here as boxes. Here, for example, the name, and then a reference to some memory location...where something is. *Points at the visualization to indicate that he is speaking of a variable name and the reference stored in the variable, which points to the heap.*

Jan Erik goes on to explain how he interprets various parts of the visualization. The Classes and Functions panels, for instance, mean that "what functions and classes we can call, create, and use" is "stored somewhere in memory". These computer-memory-related concepts are perceived to be the central content of learning in UUhistle. Unlike Lynda, Jan Erik is pleased with this:

> **Jan Erik:** At least for me, it's been [i.e., UUhistle has been] an interesting new acquaintance. Even though I've taken some other programming courses, I hadn't realized all this, how it really goes...These heaps and stuff...frames...So in this way it's been useful, this UUhistle.

Jan Erik further explains that he has learned about how objects first get created in the heap and how the computer then forms references to them. However, Jan Erik does not perceive there to be much direct use for what he has learned.

> **Interviewer$_1$:** Do you feel that it [UUhistle] has in any way contributed to your learning?
> **Jan Erik:** Yeah, I'd say so...Even though, myself, I had some earlier experience with programming, so the exercises – the coding exercises – have been easy. So I didn't learn any coding in the sense that this would tell me what to do in the exercises. But this structure *(points at UUhistle's display of memory)* became clearer. So even though in the basic course on Java had, in one of the lectures, an explanation of what the heap is, and what a frame is...they never really stuck in my mind.[11] But here I've properly figured out what data goes...what data goes where and how these things work, so this did, like, make that clear in a big way. I mean this here about how it goes, like, beneath the surface, that was really clarified.
> *Later, he remarks:*
> **Jan Erik:** From this [the use of UUhistle in CS1–Imp–Pyth], what I've come away with is how this data works in the heap. I've managed so far without that information but it's nice to know. Can't hurt. Pretty interesting.

For Jan Erik, implementation matters are an issue separate from actual programming. He feels that he knows how to program already and that learning about how the computer works does not significantly contribute to his ability to program, interesting though it may be. He does not, for instance, appear to think that knowing about references improves his ability to program. At least, he does not focus or comment on such improvement in this context.[12]

One more example from Morten explicitly illustrates how in Category B, as in Category A above, the learning gains from UUhistle are directed inwards towards UUhistle itself, or at least not towards programming in general. Morten describes UUhistle as a system that shows "what the computer does within itself, like, without necessarily showing us what it's done". During the VPS session, he first mentions that the assignments have been "useful". However, when asked to elaborate, he clarifies: it is not that the VPS assignments have been useful, but that the program animations in UUhistle have been useful for the purpose of doing the VPS assignments:

[11] Jan Erik had earlier taken another programming course in which the lecturer – the author of this thesis, as it happens – showed animations in PowerPoint that were similar to UUhistle's program animations.

[12] Jan Erik, above, does seem to imply that others might learn what to do in coding exercises from UUhistle, even though he did not. While the views Jan Erik expressed during the interview were generally typical of Category B, in this instance he expresses a view that brings to mind Category E.

Interviewer$_2$: Can you describe **how** these have been useful... You said that they have been useful somehow? Can you elaborate?
Morten: Well, it's more, like, that for the purpose of the UUhistle assignments – for doing the actual [VPS] assignments – that the examples have been useful.
Morten goes on to complain about how it is hard to see where the error is in UUhistle.
Morten: So it's, like, for the purpose of UUhistle itself that UUhistle has been useful, but I don't really know how useful it is for... coding.

To summarize, in Category B the act of learning is to learn to understand how the computer works by simulating the computer's behavior during program execution. Whether this act is meaningful for the learner depends on the learner's interest – or lack thereof – in learning to understand the computer. The objective of learning about the computer is, in this view, only vaguely linked to programming ability and is essentially seen as something separate from 'actual coding'.

17.4.3 C: VPS is perceived as giving a perspective on execution order

Kate$_1$: *To UUhistle:* Well, aren't you being careful that we do it in the correct order!

UUhistle requires a specific order of doing things. This is something that students focus on and that they have trouble with. In our video observations of VPS, students often discussed the order of simulation operations – dialogue in the vein of "first you click there, then create the other parameter", "don't create it yet, only after you have something to assign to it" is common.

Category C is an extension of Category B. As in that category, VPS is perceived to teach how the computer executes programs. In Category C, an additional emphasis is placed on a particular aspect of computer behavior, the actual execution order of programs when run by a computer. Learning through VPS is perceived as *gaining a new perspective on the execution sequence of programs within the computer*.

Otto comments:

Interviewer$_2$: ... what do you see here and what is the whole point?
Otto: We go through the program code item by item, so that we see how the computer sees it.
Interviewer$_2$: Mmhmm. *Brief silence.* What does it mean to you, that the computer sees it like that?... does it see it like that?
Otto: Well, not like that, I guess there's some ones and zeros running about.
Interviewer$_2$: Yeah.
Otto: But it, like, handles things in this order.

William also singles out order of execution as what UUhistle has given a new perspective on. He gives a concrete example and remarks that the order he has been taught by UUhistle differs from the order in which he thinks about program elements when he writes code.

William: This [UUhistle] brings a different perspective. Here the code is ready, and we view in what order the computer, like, executes it, what its way of thinking is.
Interviewer$_1$: So, what is it that you can say about the "way of thinking" of the computer based on it?
William: *Laughs nervously.* Well, it executes the commands according to what they mean in the code language... and it always starts from the right! *Laughs again.*
William elaborates, referring to assignments: he means that first the right-hand side gets evaluated, and then "what comes out from there is given a name".
Interviewer$_1$: Has UUhistle highlighted this in a different way somehow than writing your own code has?
William: Yes, yes, definitely.
Interviewer$_1$: Can you explain how it's different?
William: Well this starts with... the content. And then it's given a name. Here in UUhistle.
Interviewer$_1$: Mmhmm.

William: When I write code myself, I start by thinking that I'm going to name this thing somehow and then it's going to contain something, and then I take it from there.

Even though a new perspective on execution order can be gained through VPS, it is not necessarily clear how to make use of it. William, for instance, vaguely remarks that the UUhistle assignments have been "fairly useful, as such", but states that it feels "useless" to have both program-reading and program-writing assignments "about the same topic", especially as the program-writing assignments are "definitely more useful". Although UUhistle assignments give a new perspective on program-writing assignments, a further relationship between the two is not perceived (cf. Category F below).

17.4.4 D: VPS is perceived as learning to recall code structures

Category D describes a way of experiencing learning through visual program simulation as *learning to recall programming commands and patterns of code*. This category extends Category A.

In Category D, as in Category A, the visual aspects of visual program simulation are perceived as relatively meaningless graphical manipulation that follows the VPS system's rules and is primarily motivated by a need to complete assignments successfully. Beyond Category A, in Category D the VPS system is perceived as a platform where the student encounters example code.

Interviewer$_2$: Let's say one of your friends hasn't done any of these assignments and asks you if they're worth doing and what, like, the point of these is? What would you say?
Raymond: Well, probably that here you learn to recall these commands *(points at `print`)* a bit and...
Interviewer$_2$: Commands. What do you mean?
Raymond: That it says "print" there. And what it's supposed to look like. *Waves mouse cursor around the entire program code.* But I think those [program-writing assignments] are much more useful than these.
Interviewer$_2$: Yeah.
Raymond: So, this is, like, what it is. *Pauses.* It pays to collect every stray point. *Continues with the assignment.*
Interviewer$_2$: *Interrupts.* Yeah. So, um, you said that this is "what it is" *(small laugh)*, so I'm intrigued by what "what it is" is?
Raymond: *Eagerly:* Well, it's, like, awfully frustrating that the box won't go where I drag it, and... the usefulness... you go a bit like "does this make any sense?".

Program code is focal to this view. For Raymond, what learning potential there is in visual program simulation involves the program code (and not the graphics). Seeing the code may lead to a better ability to recall commands. Additionally, a code example can give an idea of what a program "is supposed to look like". Otto expresses a similar view:

Interviewer$_2$: How would you say, why do you have these things in your course? Does it benefit someone somehow, to have these assignments there?
Otto: Well, at least you get to see a bit of that code. Later you have to write, I think, the same kind of thing in the [program-writing] exercises. So, when you do, you don't have to completely pull it out of thin air.

Being exposed to example code gives Otto something to work with in exercises[13] This is something that he feels improves his program-writing skill.

To summarize, the act of learning in Category D is to learn to recall the individual commands or more general patterns that one has seen in example code. The why of learning is to improve one's ability to write programs.

[13]Otto, as an individual, appears to rely greatly on patterns of code when he thinks about programs. Early in the interview, he gets badly stuck in the VPS exercise because the example program does not start with (or contain) an input-reading instruction. He is "confused that the program doesn't have any `raw_input` command [...] in the code that I write, there's always a `raw_input` bit".

17.4.5 E: VPS is perceived as learning to understand programming concepts in example code

In Category E, learning through VPS is perceived as *learning to understand programming concepts present in example code*. This category is an extension of Category D. As in Category D, learning through VPS is understood to be founded on studying example programs and taking things from the examples into one's own program-writing work. There is one significant difference, however, between the categories. Whereas in Category D, little or no meaning is attributed to the visualizations, in Category E the visualizations have a purpose as clarifications of example program code and programming concepts present in the code.

> **Interviewer$_2$:** You said that you get a "big picture"... What is it that you learn? Or what do you mean by big picture?
> **Beth:** Mmm... *Thinks for a bit.*
> **Interviewer$_2$:** I ask all these terrible follow-up questions... *Laughs.*
> **Beth:** *Eagerly:* Awfully difficult to... like... You kinda learn to understand what it is that they're after. That: Okay, here we have these variables and over there we have some data, and some operators at some point, too. Because you don't really... Like, these [assignments in UUhistle] are quite easy, in a way.
> **Interviewer$_2$:** Mmm.
> **Beth:** And here we have everything explained so clearly. So, like, [in the program-writing assignments] you don't have things explained so that you **see**, instead you're just told in words what you're supposed to, like, do. And you have to divide it into these parts yourself. So, that's what I mean, like, that you get a big picture because you see the parts all at once here. *Points at the display.* You don't have to look for it and try to build it yourself like in the [program-writing assignments].
> **Interviewer$_2$:** You said there are "parts". How do you think about what those are?
> **Beth:** *Points at the main panels of the UUhistle GUI.* Well, it's like, there's three parts. An operator, like what you do with the data, which is up here. And these are variables that contain operators and the data that's inside them. And then here on the left *(points at the code)*, you see where you're at.
> **Interviewer$_2$:** Yeah.
> **Beth:** So I mean that in a way there are four parts, in total.

For Beth, the VPS assignments clarify what goes into a particular program and what the role of each piece of code is: what serves as data, what the operations performed on the data are, what the variables are, and so forth. The visualization and the program code are alternative ways of expressing the same information. Beth opines that the visualization is in fact a better medium for expressing such information in lucid, beginner-friendly manner than code is.

> *Beth and the interviewer are discussing an example program in which objects represent cars.*
> **Interviewer$_2$:** Here we have these `Car` boxes. You said they are... that they somehow clarified things?
> **Beth:** Yeah. Well, I mean that since they're visual like that, it lets you see that: okay, we have a car there and it contains this gas tank and that amount of gas. And it kinda says all this here. *Circles the mouse cursor around the Car object.* But in [the program-writing assignments] if there's some piece of code that says it, it's much harder to make sense of than this here.

Beth does not discuss, or appear concerned with, how computers execute Python programs. For her, such matters do not appear to bear on the design of the visualization or the required sequence of GUI operations. What matters more is that the VPS assignment should clearly illustrate the "parts" of each program and that it is convenient to find the GUI operations that allow her to produce this illustration. During the interview, Beth offers suggestions as to how she thinks the assignments could be made "more logical". For instance, she would like to split the heap into a part that contains elements that the user has created and another with "what is given", and she would prefer to first determine which method is called

and only then which object it is called on. In contrast to Categories B and C, in Category E neither control flow nor the computer's role in executing the program are focal. The way of understanding described by Category E sees programs as relatively static – program dynamics are not in focus – and the visualization is not seen as a step-by-step trace of what actually happens, but as an educational device that clarifies what entities are present in the program and what connects where.

As VPS assignments clarify examples and give a breakdown of program components, they can also help users learn about new kinds of components and component relationships. Beth has learned about objects.

> *Asked for a concrete topic that she has learned about, Beth brings up the last round of exercises, which introduces objects and classes.*
> **Beth:** Well, the latest is that I've – I've not done the last [program-writing assignments] on objects yet – but I took a look at the assignments in UUhistle, because I didn't have, like, any idea of what an object is... in programming.
> **Interviewer$_2$:** Mmm.
> **Beth:** So that's the kind of thing you learn to understand through it.

During the interview, Beth works through a VPS exercise in which two variables refer to the same `Car` object.

> **Interviewer$_2$:** Could you explain in more detail what it is that you get out of this?
> **Beth:** Mmm... Umm... *Pauses.*
> **Interviewer$_2$:** From this exercise, for instance?
> **Beth:** Um... Well, here you at least... Someone can tell you that you can do this kind of thing, but here you learn in a concrete way, for instance... I think the idea here is that you learn that you can start referring to something *(points at a statement `workCar = ride` in which a reference to a car object is assigned to another variable)* with another name by writing it like this.

In Category E (as in D), learning about example programs is motivated by being able to apply ideas when writing programs. In Category E, this means taking programming concepts and techniques from the examples. Beth alludes to this above as she mentions how she has noticed that you can "write it like this" to make two variables refer to a single object. When asked directly, she is more explicit about how VPS has helped her build a foundation for doing the program-writing assignments.

> **Interviewer$_2$:** You said that these give a big picture, but that the [program-writing assignments] are more difficult. Are these two connected, these assignments?
> **Beth:** Yeah! They are. *Pauses.* Isn't the idea that they are about the same topics?
> **Interviewer$_2$:** Mmm.
> **Beth:** Yeah.
> **Interviewer$_2$:** How about when you said that this gives you a big picture... Can you somehow make use of that in the [program-writing assignments], or are they, like, something separate?
> **Beth:** I think you can make use of that. What it gives you is... You can't really do them just based on these, I mean, these are quite different-looking after all, but you do get a bit about what the trick to this business is. Or at least since I haven't gone to the lectures and stuff, so otherwise it's quite hopeless to get what the trick is.
> **Interviewer$_2$:** Yeah.
> **Beth:** And it's pretty painful to start reading the handouts, too, so this [UUhistle] has been a sort of easygoing thingy for getting started.

17.4.6 F: VPS is perceived as learning programming through an understanding of program execution

Category F describes the richest understanding of learning through VPS that we found in this study. In this category, which extends Categories C and E, and therefore all the other categories as well, learning

through VPS is perceived as *learning programming via learning about program execution*. This view is more than the sum of the other categories in that not only is VPS perceived to be about the computer (cf. Category C) and about learning programming concepts that can be applied in one's programming work (cf. Category E), but a connection is perceived between these aspects: learning about implementation concepts and what the computer does is seen as improving one's ability to program. This improvement concerns both the ability to understand programs and to write them oneself.

Sue sees UUhistle's visualization as a reflection of computer memory and what the computer does. She is enthusiastic about how useful the system is for understanding program code, partially because it makes her think about execution in detail and allows what Sue calls a trial-and-error tactic for exploring and getting feedback.[14]

> **Sue:** Here, you have to think a lot about what it [the code] does. And obviously you do get things wrong quite a bit, too. So it becomes a trial-and-error tactic at that point... in this UUhistle... game. When you have to look at code without UUhistle, to understand it, it's difficult, because there the trial-and-error doesn't... I mean, UUhistle tells you pretty well what... like, how it goes. But when you look at [just] the code, you don't see it in UUhistle's graphical way, it's hard to understand, because you can't make sense of the code as easily.

Sue finds it easier to understand programs in UUhistle. Moreover, she sees UUhistle not merely as a tool for examining programs but as a learning aid. She is quite explicit about this:

> *Sue is in the midst of discussing why immediate feedback is a good thing.*
> **Sue:** ... this [UUhistle] has been made so that you learn, so it [UUhistle] must give it [feedback about errors] right away. And then you start learning from... when you think about what went wrong.

During the interview, Sue works on a program that recursively computes a factorial. Sue has never seen a recursive program before, but learns to understand it quite well during the interview. She attributes this learning to the view given by UUhistle and the implementation concepts shown, such as the call stack.

> **Sue:** Especially now that we have... these functions inside functions... it's difficult, at least for me, to understand how... inside a function you call some function again, with a different value, for instance... so in what order does this – in principle, the computer – do things? And at which stages do you process the numbers in which ways? That is still a little bit fuzzy.
> *The topic comes up again later in the interview.*
> **Interviewer$_1$:** So, if I understood right... you feel that you get something different... that UUhistle helps you understand in a different way what you've seen elsewhere?
> **Sue:** Yeah! And right here, for instance, in this example that we did... that makes it a lot clearer... the function calls that are within one another... basically, you get a clear sense of order. From the code, you don't get, like, any sense of how long it works on it. That we find this `return` *('underlines', with the mouse, the code* `return fact(n-1)`*)*, and then the function is called yet again. *Quickly 'draws circles' around the recursive function's code with the mouse cursor.* So here *(motions at the call stack)* it's, so to speak, illustrated in pictures, and it makes it way clearer, how it works. That these... functions that haven't been executed until the end yet, they stay here in a queue waiting, as it were, until new values comes from here *(points higher in the stack)*. And those in turn fetch – call – the function.
> **Interviewer$_1$:** Mmm.
> **Sue:** In a way, the name "call stack", too, says... makes it clearer how the functions behave... in relation to each other. That they stay waiting there and, in principle, we always have only one function running. And the others wait in the meantime. This example clarified that really well, I think.

[14] What Sue does as she works on VPS exercises involves a lot of thought and is a far cry from what we would call a naïve trial-and-error strategy. More on Sue in the next chapter.

Sue finds that what she has learned in UUhistle is useful for understanding programs outside UUhistle as well. She points out that learning to understand code in UUhistle is useful "because if you don't understand it here, on any level, then you definitely won't understand it in the code, how it goes."

Elizabeth is another student whose comments serve to illuminate Category F. Elizabeth says UUhistle helps her learn about "what the things really are that you write" and "what structure it actually works on", instead of just learning "how to make it work". She sees the visualization as "memory locations" within the computer. What Elizabeth learns in UUhistle is something that she feels she can take to her own program-writing work outside UUhistle. Having, like Sue above, just visually simulated a recursive program, Elizabeth suggests that her chances of writing a recursive program herself have improved. She envisions that this learning could transfer to writing recursive programs in other contexts in the future.

> **Interviewer$_1$:** What did you get from doing this exercise, generally speaking? What would you say you have accomplished?
> **Elizabeth:** Well, you do notice at least that there can be quite a few of those, um... the same function, when it gets called many times in the code... that it forms like... *(laughs)* in layers, that structure. And then it starts coming back... like a snowball or something like that. Maybe it's now easier to understand how to... if I wrote code like that myself... Of course I'd have to know everything else as well, but I mean, um, I'd know how the computer works on it.

17.4.7 The categories connect logically

A breakdown of our outcome space in terms of the structure of awareness (Section 7.1) is shown in Table 17.2. The table provides a systematic view of the categories of description that form the outcome space and expands on the logical relationships between them. In our outcome space, as is normal in phenomenographic results, the richer categories of the outcome space are extensions of simpler ones (as Figure 17.6 on p. 281 shows diagrammatically). In other words, some of the categories describe ways of understanding that subsume simpler ways of understanding learning through VPS.

Figure 17.6 illustrates how our outcome space has two 'branches' that extend the simplest category A and merge at the richest category F. The 'blue branch' formed by categories B and C extends the superficial understanding of Category A by emphasizing the actual operation of the computer as it works on programs. The 'red branch' of D and E extends Category A in a different direction. It emphasizes the role of VPS as a platform for studying example program code (in Category D) and the concepts present in examples (in Category E). Category F relates these two branches to each other: it is understood that learning about computer behavior serves a purpose in understanding programs and programming.

Another way of looking at the categories further highlights how the categories form a whole and how the other categories are limited relatives of Category F. Consider first the following statement, which summarizes some of the main points of a Category F view:

> To learn through visual program simulation is to use a visualization to study example programs, to learn what happens within the computer as it works step by step through the example code, and thereby to improve one's programming skills.

Each of the simpler categories has its primary focus on a different aspect of this statement, as indicated by the emphases below.

> Category A:
> To learn through visual program simulation is to ***use a visualization*** to study example programs, to learn what happens within the computer as it works step by step through the example code, and thereby to improve one's programming skills.

> Category B:
> To learn through visual program simulation is to use a visualization to study example programs, to learn ***what happens within the computer*** as it works step by step through the example code, and thereby to improve one's programming skills.

Table 17.2: Qualitatively different ways of perceiving what it means to learn through visual program simulation

Categ.	**Act of Learning** (what it is to learn)			**Why of Learning** (a.k.a. indirect object)		
	Internal horizon (parts involved and their relationships)	**External horizon** (backdrop)	**Referential aspect** (to learn through VPS is to do what)	**Internal horizon** (motivating situations)	**External horizon** (backdrop)	**Referential aspect** (what is the use of the learning)
A	Program code, visual elements, operations on the visual elements, rules set by the VPS system. Vague relationships between these components. Visual elements exist to be manipulated. The student's role is to read code and perform these manipulations.	Coursework and the VPS system.	To learn to perform manipulations on graphics as per system rules.	VPS assignments.	Coursework.	To fulfill assignment requirements and gain points.
B	As above. Additionally: The internal horizon encompasses the implementation of the programming language within the computer, and related concepts such as memory and control flow. The visualization is seen as a representation of what the computer does. The student's role is to simulate the behavior of a computer during a program's execution.	As above.	To learn, by taking the computer's role, 'under-the-hood' knowledge about how a computer works as it executes programs.	VPS assignments, and possibly other situations where under-the-hood concepts are useful.	Primarily coursework.	As above, and to have under-the-hood knowledge about programming.
C	As in Category B. Additionally, emphasis is placed on execution order and how the student's role is to use the visual elements to indicate what the actual execution order is.	As above.	As in Category B, and especially to develop a new perspective on programs that highlights the order in which programs are executed.	As in Category B.	Primarily coursework.	As in Category B.
D	As in Category A. Additionally, the student's role is to study given example code.	As above.	As in Category A. Additionally, to memorize commands and patterns present in example code.	VPS assignments and other programming situations.	Contexts in which one programs (e.g., coursework).	As in Category A, and additionally to be better at writing programs.
E	As in Category D. Additionally: The visualization is perceived to be an educational representation of a program: what the code consists of and what entities are present in the program. The student's role is to study a given example and to use the visual elements to produce an illustration of how its parts connect.	As above.	As in Category D, and additionally to learn to understand programming concepts present in example code.	As in Category D.	As in Category D.	As in Category D.
F	As in all the categories above. Additionally, under-the-hood implementation concepts and programming are perceived as being related to each other, and simulating the computer's behavior is perceived to be relevant to learning to program.	As above.	As in all the categories above, and additionally to learn to program better by understanding how computers execute programs.	As in Category D.	As in Category D.	As in all the categories above, and additionally to be better at understanding programs.

Category C:
To learn through visual program simulation is to use a visualization to study example programs, to learn what happens within the ***computer as it works step by step*** through the example code, and thereby to improve one's programming skills.

Category D:
To learn through visual program simulation is to use a visualization to study example programs, to learn what happens within the computer as it works step by step through the ***example code***, and thereby to improve one's programming skills.

Category E:
To learn through visual program simulation is to use a visualization to ***study example programs***, to learn what happens within the computer as it works step by step through the example code, and thereby to improve one's programming skills.

Our exposition of phenomenographic categories of description in this way is inspired by Marton and Booth (1997, pp. 18–19, 146–148). Accentuating the foci of the simpler categories in the context of a statement corresponding to the richest category allows us to see something of how the simpler categories represent *partial* understandings of the phenomenon under investigation.[15] Further, the way that some categories are more complex extensions of others allows us to judge the richness and quality of different ways of perceiving the phenomenon. "We are able to make a normative judgement as to which is best and which is worst, but that does not change the fact that they are indeed ways of understanding it [the phenomenon]" (Marton and Booth, 1997, p. 19). The existence of these logical relationships between categories gives credence to the idea that they are ways of understanding a single phenomenon rather than different phenomena. Being able to judge the quality of the ways of perceiving learning through VPS provides us with a foundation for discussing the significance of the categories for learning and teaching (see Sections 17.5 and 17.6 below).

Connections to SOLO

We note in passing that parallels can be drawn between our results and the SOLO taxonomy (Chapter 2). The author of the thesis remarked in Section 7.6 that phenomenographic outcome spaces tend to resemble SOLO. Ours, for one, does. One way to interpret our result in terms of SOLO is that A is a unistructural one where the concrete visualization is the only aspect (or 'structure') present. Categories B through E describe various multistructural ways of understanding, and F a relational one in which a connection is drawn between learning about computer behavior and learning programming. Our outcome space is considerably more nuanced than SOLO, however. In SOLO, meaningful connections between structures 'appear all at once' at the relational level. In a phenomenographic outcome space such as ours, even the partial ways of understanding often involve an understanding of *some* but not all relations between aspects (e.g., in Category B the visualization is related to execution within the computer, but not to learning programming).

17.5 Richer understandings of VPS mean richer learning opportunities

As we turn to discuss the significance of our findings for learning programming, it is good to remind ourselves of the nature of phenomenographic results. Phenomenographic research (see Chapter 7 and the beginning of this chapter) investigates the relationships between experiencers (people) and phenomena. The results of our study, for instance, tell us something not only about students but also about what the phenomenon of learning through VPS is like. People's understandings of the phenomenon depend on

[15] We are not claiming that Category F represents a complete understanding of the phenomenon but it is the richest way of understanding uncovered in our study.

what the phenomenon is in reality, and conversely, the phenomenon of learning through VPS is defined by what people make of it – it is constituted by the different ways of perceiving it. The two-way relationship between experiencers and phenomena is inseparable.

What matters most for learning purposes is not what learning through VPS is as an ideal, but what students make of VPS for themselves (or what VPS makes of itself for them). This is what our outcome space is about.

Our findings are encouraging and sobering at the same time. The good news is that it is possible for learners to experience VPS in a rich way that enables them to learn programming through it. The bad news is that there are a number of limited – and limiting – ways of understanding, which must be addressed in order for VPS to be useful in practice to as many people as possible.

A quick read through our results might suggest to some readers that VPS is liable to be problematic for most students, as there are many different ways of understanding it poorly. However, being purely qualitative, our results do not say anything about the frequency of the different partial understandings in student thinking. Neither should we conclude that a similarly sized but randomly selected group of students is likely to exhibit all the difficulties highlighted by our study. Nevertheless, we must take note of the existence of these categories and the difficulties they bring to the fore.

Our findings agree with what has been said about the use of visualization in education in general: a visualization can help but it is no miracle cure, and will not work 'just like that'. This also applies to the highly interactive form of program visualization that is VPS. There are multiple qualitatively different ways to relate to a visualization, and not all of them are equally conducive to learning. For a space of learning to open up for learning programming, it must first be possible for the learner to perceive the visualization in a rich way. Our categories illustrate how learners have to experience a number of key insights before effective learning through VPS can take place.

We will now comment on the specific categories and their relevance to the intended goals of VPS.

17.5.1 Limited understandings limit the potential of VPS

Consider Category A. If one currently perceives learning through VPS merely as graphical manipulation of visual elements, and learning through VPS merely as learning to perform such graphical manipulations, then performing graphical manipulations is what one learns through VPS. One may eventually learn to perform many of the correct manipulations with a reasonable degree of success, but one is unlikely to learn much about programming.[16] In this case, the *lived object of learning* (Marton et al., 2004) – that is, what the learner perceives as the capability one is striving towards – is but a scratch in the surface of the *intended object of learning* – the capability the teacher hopes that students will develop. If one does not get past a Category A understanding of learning through VPS, VPS is a waste of time. Affective issues present an additional challenge: a student who currently only experiences VPS as in Category A is likely to be frustrated and even angry at having to perform tasks perceived as difficult and meaningless.

In Category B, VPS is seen as being about implementation concepts irrelevant to practical programming. If one is interested in these concepts (as Jan Erik was; see Section 17.4.2), then perhaps one will learn about them, reflect on them, and remember them, and may yet connect them to programming practice later as one's understanding develops. If one does not ascribe any intrinsic value to implementation concepts, then whatever knowledge is learned is liable to become context-dependent, fragile, and soon forgotten.

The existence of Category C suggests that execution order is an aspect of program dynamics that is particularly emphasized in VPS. It is possible that the interactive nature of VPS, and the requirement for the student to decide what happens when, direct the user's attention to ordering more than other program visualizations do. Mistakes with the ordering of execution steps are not infrequent as novices work on VPS exercises, which may further highlight this aspect. From a point of view of learning programming, a Category C understanding is not a great deal better than a Category B one.

Category D describes a way of understanding that is little better than Category A from a practical point of view. Although VPS does present a way of encountering example code, it hardly gives good value for effort if the interactive simulation aspect is ignored or seen as meaningless.

[16] It is possible to learn some useful things inadvertently and without noticing, but we see no reason to believe that VPS is a particularly cost-effective catalyst for such unconscious learning.

Category E describes a more productive way of understanding. A focus on programming concepts allows for knowledge to be transferred from VPS to programming practice. This makes VPS a meaningful activity that more readily results in deeper learning of programming. That said, a Category E view is limited. Ignorance of the regulatory 'reality' of the computer leads to confusion about the particular simulation steps involved. The visualization is likely to be seen as excessively complicated. When simulation steps are seen as fairly arbitrary, they do not invite reflection on program dynamics.

Of the categories we uncovered in this study, we consider only Category F to describe a satisfactory understanding of learning through VPS. Only in this category is a clear link present between learning about the computer and learning programming. Because of this, we feel that in order for VPS to be a successful learning activity in introductory programming courses, users of VPS need to be able to adopt a Category F view as early as possible. Category F is also desirable from an affective point of view: a task perceived as a meaningful part of the programming course is more likely to be embraced by students.

17.5.2 VPS is only worthwhile when students understand it in a rich way

How do the goals of VPS look in the light of our qualitative evaluation? Nearly all of the claims and hypotheses set down in Chapter 14 are predicated on students being able to perceive VPS in a fairly rich way.

VPS does not help students perceive programs' existence as both static and dynamic unless VPS is perceived at least as in Category B.

VPS will not result in meaningful cognitive engagement with a visualization as long as the student perceives his actions as essentially meaningless, as in Categories A and D. Misconceptions will be left unaddressed by the visualization unless meaning is attributed to it.

VPS will not help students to construct and ingrain a viable model of a computer unless the visualization is understood to represent a computer as in Categories B, C, and F. Neither will VPS be of significant help with difficulties concerning program state.

VPS will not succeed in teaching much about programming concepts without at least a Category E understanding, or preferably a Category F one.

VPS examples will not serve as worked-out examples for program writing unless a connection is perceived between viewing example code in a VPS context and program-writing tasks (as in Categories E and F, and, to a limited extent, D).

If VPS is understood to be only about the computer and implementation-level issues, then it will not help students develop the ability to form program models as they read programs; an understanding akin to Category E is needed.

Even while a student thinks about VPS as in Category A, they are likely to form some sorts of schemas that help them perform the required GUI operations. However, they will not form the sorts of schemas that are useful for learning to program unless the relevance of VPS to programming is understood (as in Categories E/F). Only when the student thinks about VPS in terms of the underlying concepts will they form schemas that involve the semantics of code constructs and serve as the building blocks for higher-level schemas.

VPS may invite students to directly manipulate the important but usually tacit conceptual content of programming that is program dynamics, but the invitation is likely to go unanswered unless that content is perceived to be present in the VPS system, and relevant to learning to program, as in Category F.

VPS is unlikely to be a helpful way to practice program tracing, or program understanding in general, unless a Category F understanding is achieved.

It is clear that a VPS system is no free ticket to success for the CS1 teacher. Our evaluation highlights the need for – and suggests strategies for – good teaching about VPS.

17.6 The relevance of VPS needs to be taught

Our findings have implications for introductory programming teachers who use VPS and for the designers of VPS systems. For VPS to work for as many students as possible, an early effort should be made to help

the students develop a sophisticated understanding of what they can get out of program visualizations and VPS exercises. VPS must be carefully integrated with other teaching and learning activities.

Getting to the students early is important to avoid bad first impressions, negative emotions, and waste of time. We recommend that when a VPS system is used in a course, it be introduced early on, preferably in the first or second week. At the same time, teachers should try to ensure that students learn to experience the system in a rich way, and foster open-mindedness towards what can be learned through VPS. Ideally, students would get to engage with the system themselves immediately, supervised and guided by a teacher (although this may be impossible in many large-class scenarios). When possible, teachers should try to find out how individual students think about VPS and help the students improve.

Our outcome space highlights aspects that early guidance should focus on. There are a number of categories and aspects to consider; however, we have tried to distill the problem into two interrelated teaching challenges:

1. Teachers should *help students perceive a meaning behind the visualization*. This means coming to gain the insights described by the 'blue branch' of our outcome space, in particular the key notion that the visualization represents computer behavior.

2. Teachers should *help students come to see a purpose to the learning* by making them understand that these insights are related to programming practice. This corresponds to becoming able to view learning through VPS as in Category F.

These two main pedagogical challenges are, we conjecture, not limited to VPS only. None of the students we interviewed appeared to find the program visualization itself to be meaningful and useful while at the same time considering the interactive VPS activity to be incomprehensible or pointless. While reading the following recommendations, the reader may agree with us that much of the advice we give here could also be applied to forms of educational program visualization other than VPS.

More broadly still, the literature of education and psychology agrees widely with our results in that stressing underlying principles and providing motivating contexts for learning are highly important for meaningful learning. Our findings concretize these issues in the context of programming education and VPS, and document the conceptions learners have of the relationships between the VPS system, the visualization, the underlying programming concepts, and learner motives.

17.6.1 Teachers should help students perceive meaning in the visualization

For students to discern – within the phenomenon of learning through VPS – the critical aspect of computer behavior, they need to be made focally aware of the fact that the computer behaves in a certain way as a program is executed. Furthermore – and this is probably the difficult bit – teachers must find ways to relate this critical aspect to the visualization aspect of the phenomenon. In variation-theoretical terms, different visual elements and simulation steps are values along one dimension of variation, and correspond to different computer behaviors which are values along the other dimension. The teacher can tackle this challenge by underlining these correspondences and creating learning situations that encourage students to reflect on them. This can be accomplished through classroom discourse, materials (texts and program examples), and the VPS system itself.

Teachers need to find ways to draw students' attention simultaneously to visual elements and their meanings. For example, students should be expressly taught that the rectangle representing a frame corresponds to a concept that is relevant to the implementation of the programming language within the computer. Teachers must not assume that the visualization will be obvious to novice programmers nor that the interactive nature of VPS will be enough for students to discover the meaning of the visualization unassisted.

Teachers can explain the visualization in terms of what the computer does, and what the computer does in terms of the visualization. The student's task can be explained as taking on the role of the computer.

Teaching should emphasize that although it is possible to visualize what the computer does in different ways, the visualization is not arbitrary, nor are the execution steps that the student is expected to follow during VPS.

Class discussion and learning materials can relate new programming constructs to what is needed to execute a program that uses those constructs. For instance, the topic of frames can be approached through the observation that any executor of programs needs to keep track of the variables (among other things) that pertain to function or method calls. The role of the visualization (and VPS) is to illustrate what techniques the computer uses for this purpose as it runs programs.

Example selection is key to the success of VPS exercises and program visualization in general. Using simple programs as examples allows cognitive load to be managed while singling out specific aspects of execution. However, minimal examples alone are often not enough to draw attention to all the critical variation and to justify the complexity of the execution model presented. For instance, if all the variables in the visualized example programs have different names, one of the main reasons why the computer uses multiple frames to keep track of state will not be readily appreciable (and requiring students to create frames in VPS may seem at least equally pointless).

Another example. The order in which parameters are evaluated may seem to the learner largely an arbitrary choice of the VPS designer (cf. the 'red branch' of our outcome space). If all the parameter expressions are simply literals, variable names, and arithmetical expressions with no calls and no 'side effects', then evaluation order might indeed not matter. It is important to also visualize example programs where order does matter. This can lead to a discussion of how the computer needs (and the programming language provides) an unambiguous definition of how to execute the program. Students can be invited to ponder interactively whether or not an alternative order or simpler usage of visual elements could work. (Students may also volunteer such alternatives, as Beth did; see p. 288.) Through such discussions, students can be led to conclude that the visualization is a way of illustrating the specific principles that the computer follows so that it can work on the general case, and not just a particular simple kind of program.

Chapter 18 explores what students do during VPS sessions. We will have more to say there on the kinds of pedagogy that can encourage students to discern and engage with the conceptual content of VPS.

17.6.2 Teachers should help students see a purpose to VPS

To encourage transfer from VPS to programming, teachers must help students experience a connection between computer behavior and useful programming activities.

An important starting point is the way VPS is initially described to students. VPS should be introduced as a tool for helping the student learn about programming concepts by looking at how the computer deals with programs. Teaching should stress the idea that the visualization gives an execution-time perspective on programs, and that adopting such a perspective is frequently useful to the programmer as they reason about programs.

Novices often focus strongly on the programming language they use. From the start, it should be stressed that VPS can help them understand the language and therefore how they can use the language in their own programs.

Some teachers – the author of this thesis, for one – like to explain to their students some of the pedagogical strategies behind course design, occasionally with reference to learning theory. In the case of VPS, students could be told that the interactive nature of the visualization is something that is intended to encourage them to pay careful attention to the visualization and reflect on it, so that they will become better programmers.

Above all, teachers should go to some trouble to demonstrate explicitly and concretely how understanding program visualizations can help the students read and write program code. For students to experience the usefulness of learning about the execution model, they need to be placed in situations where program animations and VPS exercises interact with other programming tasks in a meaningful way. In-class activities and open labs can both be designed to emphasize the way the visualization can help answer pertinent questions. Here are some examples of potentially useful practices (not all of which

involve VPS).[17]

- The teacher uses UUhistle (or a similar system) in class in program animation mode to explore how example programs work. The visualization serves as an illustration of useful examples. The examples are carefully chosen so that the programs are difficult to understand while their execution is hidden but easier when it is explicit. VPS might also be used in class so that students vote on key steps, and incorrect answers are also explored and explained (cf. Pears and Rogalli, 2011).

- Example selection draws on misconception catalogues so that the example programs encourage students to experience fruitful cognitive conflict and detect their misconceptions. When possible, teachers engage students in conversation about the mistakes they make during VPS. Teachers can explicitly warn about common misconceptions in class, and use the visualization to demonstrate their non-viability.

- Students encounter example programs that work in mysterious or unexpected ways that cannot be explained without a better understanding of related concepts (e.g., references, parameters). UUhistle is then used to figure out why the program works as it does.

- As a special case of the above, UUhistle is used to find a bug in an example program, which can then be fixed.

- Students use (the full build of) UUhistle to visually debug their own programs.

- VPS exercises are explicitly referred to in other assignments. This can take many forms. For instance, a program-writing assignment may refer to an earlier VPS exercise as a necessary prerequisite, or a VPS exercise may be embedded into a larger assignment, or it may be mentioned as a potential learning aid to be used in case the student has trouble with a program-writing task. Crucially, the purpose and goals of the VPS exercise in relation to the other assignment are made clear so that students know why they are doing the specific exercise.

- Visualization-based assignments feature 'planlike' example programs that have explicit high-level goals.

Encountering just one or a few early cases where a concrete benefit is gained from understanding the visualization may greatly affect students' perceptions of the VPS system, and motivate them to adopt a deeper approach as they work on future VPS exercises.

17.6.3 We have already incorporated some of our advice into UUhistle

Some of our recommendations on teaching about VPS can be incorporated into a software tool such as UUhistle. We have already begun work on some of them. As soon as the first drafts of our outcome space emerged, we reprioritized the items in UUhistle's feature requests list, and added to it, on the basis of our results. Broadly speaking, we switched priorities from seeking greater coverage of language features and other technical improvements towards features that encourage students to seek meaning in the visualization and enable them to genuinely benefit from the tool. The main differences between the prototype that the students in this study used and the current version of UUhistle from Chapter 13 were outlined in Section 16.4.2. The following recent features were partially inspired by the results we have presented in this chapter:

- the Info box and its associated explanatory texts, which seek to draw the learner's attention to the conceptual content of the visualization and invite exploration of programming concepts;

[17] We may reflect on the suggestions listed here against the context of the CS1–Imp–Pyth course that we studied. In that course offering, there was some use of UUhistle's visualization in lectures, but for the most part, the integration of UUhistle and VPS with other aspects of the course was limited (see Section 16.4). In effect, it was largely left to the students to perceive meaning in the visualization and to relate it to other assignments and programming in general.

- the "What is this?" menu choices accessible through context menus (see Section 13.1.3), which serve a similar purpose;
- the features that attempt to draw the user's attention to points of interest or to directly address specific misconceptions through textual materials (see Sections 13.1.3, 13.4.2, and 15.3);
- various small additions and quality improvements towards the goal of making the full build of UUhistle as usable and robust as possible so that students can use it to debug their own programs. Further improvements in this vein have high priority in UUhistle's development at the present time.

Such additions, useful though they may be, are unlikely by themselves to be sufficient to address the issues highlighted by this chapter. Presenting VPS in the right way in teaching and integrating it well with other course materials remains, we believe, of the utmost importance in order for students to make the most of VPS. This requires, at least at present, effort from human course personnel.

17.7 There are vague spots in our analysis

> *Finding patterns is one result of analysis. Finding vagaries, uncertainties, and ambiguities is another.* (Patton, 2002, p. 437)

We comment briefly on some of the alternative analyses we discarded in favor of the one presented, and a couple of vague spots in our results.

Our outcome space as presented has two branches. In contrast, the early versions of the outcome space were clean hierarchies in which each category extended a single other category. These early outcome spaces were problematic, however: it was difficult to determine which of the categories 'in the middle' were extensions of which other categories. Our data eventually led us to the current bifurcate outcome space as it appears to be possible to perceive learning through VPS as either computer-related or program-example-related while unaware of the other 'branch' (cf., e.g., how Lynda and Beth discuss their understandings above).

It can be questioned – we did – whether there is a genuine qualitative difference between Categories A and D. Category D adds to Category A merely the realization that as one does VPS exercises, one ends up seeing program code, and some aspects of the code may end up being remembered. This realization is arguably very obvious. It is possible that even the simplest way of understanding the phenomenon encompasses this realization and that Categories A and D should be merged into one category in which all the other categories are rooted. However, our data does suggest that ways of understanding exist that involve no consideration of VPS as a platform for example code. At least, such consideration is not necessarily present in the dialogue of novice programmers, despite in-depth probing.

The justification for the existence of Category C as a separate category can also be questioned, the alternative analysis being that execution order is just one aspect of computer behavior and Categories B and C are qualitatively similar. We did, however, feel that our data tentatively supports the existence of Category C as a category of its own, given the great emphasis that 'actual execution order' appeared to have in students' thinking and dialogue. That said, Category C is arguably not quite as solidly founded on evidence as other parts of our outcome space.[18] Students do talk about execution order, they do focus on it as they solve problems, and they do find it difficult. Two students (William and Otto; Section 17.4.3) highlighted execution order as what UUhistle gives them a new perspective on. This is a vague area that further research and richer data could illuminate better. Our pedagogical recommendations are not predicated on the existence of Category C separately from Category B.

Category E features another issue that we wish, in hindsight, that we had richer data on. This category is entirely based on a single interview. In particular, the role of program dynamics in this category is somewhat vague. What we can say from our data is that the interviewee, Beth, did not focus or comment on the computer's role in program execution, and did not appear to relate the changes in the visualization to events in the computer. However, our data does not give a clear picture of just what Beth

[18] All our categories are, to an extent, based on conjecture, as are, in our pragmatist view (see Section 16.2), most or all scientific results.

thought about the dynamics of VPS. Our 'best guess', based on what we have, is that she considered the user's actions to be largely arbitrary operations (cf. the tools available in a drawing program) which she uses to produce a visualization of program structure.

Finally, our outcome space does not precisely define, for every category, how widely applicable what is learned during VPS is expected to be. Consider Table 17.2. When it comes to the internal and external horizons of the why of learning (the second- and third-from-right columns), several categories of description are vaguely phrased. In particular, the phrase "programming situations" appears without a clear definition of what it means or a specification of the larger scope for such situations. Here, our data does not allow us to be more specific. The existence of this vague spot is explained by the fact that students' conceptions of programming in general were not focal to our study – we considered programming itself to be a separate, although related, phenomenon. However, we can compensate for the vagueness by relating our outcome space to earlier work which has problematized what programming means to novices. Consider, for instance, the work of Thuné and Eckerdal (2010), described in Section 7.5. According to that study, programming may be perceived (among other things) as merely the production of program texts, or as a skill for solving problems in everyday life. These two views lead to two entirely different meanings for the phrase "programming context". Systematically relating, on a collective level, two outcome spaces of understandings of two related concepts appears to us a theoretically and practically challenging general problem that is beyond the scope of the present work. However, we can make the generic observation that on an individual level, how a student views VPS is related to how they view programming in general, and a richer understanding of programming contributes towards envisioning a wider variety of situations where one may apply what one learns during VPS.

Chapter 18

We Explored What Happens During VPS Sessions

This chapter reports empirical work conducted by the author of the thesis in collaboration with Lauri Malmi. We explored the question:

> *What happens during visual program simulation sessions?*

Like the phenomenographic study of the previous chapter, the work we present in this chapter is exploratory. Visual program simulation is a new instructional approach about which not much is known. We sought a rich, empirically founded description of what students do as they work on VPS exercises. Such a description can give practitioners a better sense of what goes on and can inspire further research questions.

The above research question is broad. In this work, we focused on two subquestions, the second of which is an open-ended one:

1. *What informs students' choices of simulation steps?*
2. *What other interesting episodes can we observe?*

In Section 18.1 below we describe our research methods. Section 18.2 presents our main findings, which are complemented by some quantitative results in Section 18.3. In Section 18.4, we consider the implications of our findings for VPS system design and pedagogy.

18.1 We analyzed recordings of students

18.1.1 The data came from observations and interviews

We used data from both observations and interviews. We surmised that a blend of more 'natural' data with interviews, which allow probing questions from the researcher, would give us a rich idea of what happens during VPS session.

Observations

We collected videos of 41 pairs of students working on two VPS exercises in UUhistle. The students had been asked to discuss what they did as they solved the exercises.

The students were volunteers who had been solicited with an announcement on the course web site. They were roughly seven weeks into the spring 2010 offering of CS1–Imp–Pyth (see Section 16.4). All of the 41 student pairs featured at least one student who had used UUhistle in previous assignments; in most pairs, both had. The students received a small number of assignment points for taking part.

This observation data was collected as a part of a broader research setup, which we describe in the next chapter. (The pairs formed the VPS group in the controlled experiment reported in that chapter.) In short: the pairs were given one program animation and two VPS assignments, and their task was to

solve the assignments using the animation and/or any other material they wished to make use of. The programs featured lists, reference assignment, and parameter passing.

The code for the assignments is in Appendix C.

The topics of lists and references were still new for the participants. They had had a lecture on these topics (which some had and some had not attended), and some of the students had already done one or more program-writing exercises featuring lists when they participated in the VPS session. None had seen any visualizations of lists in UUhistle before.

Our videos consist of an audio track and a screen capture. One of the videos is missing a substantial segment as the pair inadvertently switched off the recording. The other recordings contain the entire VPS session.

Interviews

In addition to the observation data, we used the 11 phenomenography-driven interviews from Chapter 17. The interviews were conducted for the primary purpose of studying people's ways of experiencing, but also featured material that gives insights into what happens during VPS sessions.

18.1.2 We examined the data using qualitative content analysis

Our analysis process can be described as data-driven qualitative content analysis, given a sufficiently broad definition of this ambiguous term.

Qualitative content analysis

In a much-read guidebook on content analysis, Krippendorff (2003, p. 18) defines the term as "a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use". Krippendorff further "question[s] the validity and usefulness of the distinction between quantitative and qualitative content analyses", as "ultimately, all reading of texts is qualitative, even when certain characteristics of a text are later converted into numbers" (ibid., p. 16).

Our present work is not content analysis in Krippendorff's sense. Our work is not characterized by the strict requirement of replicability (see Chapter 16), nor do we use most of the specific techniques outlined in Krippendorff's book.

Our usage of the term *content analysis* matches that of Patton (2002), according to whom "content analysis is used to refer to any qualitative data reduction and sense-making effort that takes a volume of qualitative material and attempts to identify core consistencies and meanings" (p. 453). Our work is qualitative not only in the sense that we look at the different qualities present in our data, but also in the sense that the "core consistencies and meanings" are the main result of this work; we are not interested, here, in frequencies or distributions.

We can further characterize what we did as cross-case analysis (as opposed to case analysis of specific pairs). We looked for salient themes in the data and sought to identify instances of student behavior and thought related to those themes. Students' choice of simulation steps was one theme of interest that we had already decided to look for prior to the start of the analysis.[1]

We also hoped to be able to document concrete instances of learning through VPS. Other themes we let emerge from the analysis so that we could comment on those aspects about which our data appeared to 'speak'.

Researcher roles

The author of the thesis was the lead researcher in the work described in this chapter, and had the main responsibility for all aspects of the work. Lauri Malmi critically reviewed the resulting analysis (but not the original data, beyond what is cited) and gave feedback.

[1] To be more precise, we had initially hoped to categorize strategies or processes that lead students to choose a simulation step. However, the analysis suggested that it was difficult to found such a categorization on the data we had. This is why we reformulated one of our research questions during analysis so that we looked instead at what kinds of information students use as they make choices.

Kimmo Kiiski and Teemu Koskinen supervised the sessions in which the observational data was collected (more on the sessions in Chapter 19).

Analysis process

> *The most frequent form of interview analysis is probably an ad hoc use of different approaches and techniques for meaning generation.* [...] *no standard method is used for analyzing the whole of the interview material. There is instead a free interplay of techniques during the analysis.* (Kvale, 1996, pp. 203–204)

Kvale (1996, p. 181) points out that, in qualitative analysis, one must not overemphasize standardized techniques. Instead one must be ready to "go beyond method and draw upon the craftsmanship of the researcher, on his or her knowledge and interpretive skills". In this spirit, we conducted the analysis in a fluid, non-algorithmic manner. Nevertheless, we may roughly describe how the analysis process unraveled as three main phases.

During the first phase, the lead researcher watched the videos carefully, searching for anything of interest, and watching out for emergent themes and possible categorizations. During this process, the researcher made notes paraphrasing what happened during interesting episodes in the recordings, and iteratively sketched out categorizations of the justifications that students gave for VPS steps. Each video was considered against the background of the ones already seen, comparing and contrasting. A categorization gradually evolved. The researcher sometimes went back to an earlier recording to check how it matched a tentative categorization. The first phase resulted in a draft categorization of the types of information students use when choosing simulation steps, and a collection of notes on other potentially interesting episodes in the data.

During the second phase, the lead researcher sorted the notes collected during the first phase so that they corresponded to a few themes of interest. He reflected on these, trying to capture the significance of each episode. This phase again involved going back – at whichever point it seemed useful – to the original recordings for more detail. The second phase produced a draft of the results described in Section 18.2 below.

The third phase of the analysis was a second sweep through most of the recordings. While watching the videos, the lead researcher looked for any episodes of interest in the data that had previously been ignored or that were in conflict with what we intended to report.

Overall, our analysis process can be seen as gradually moving from 'open coding' and inductive logic towards deduction and a confirmatory phase during which we took the findings back to the data for a final examination of their appropriateness (this is in line with Patton, 2002, pp. 453–454).

Because of practical concerns – lack of time/money, combined with a large amount of data and the need to consider not only speech but also the state of the VPS GUI – we only transcribed verbatim those parts of the observational data that are quoted below to illustrate our results.

18.2 Various noteworthy features of students' VPS work emerged from the analysis

The four subsections below report our results. In order, they deal with:

- the kinds of information students use when choosing a simulation step,
- different kinds of episodes leading to learning,
- other pedagogically interesting patterns of behavior, and
- problem spots and mistakes by students.

18.2.1 Students use different kinds of information when choosing simulation steps

We identified five kinds of information, summarized in Table 18.1, that students use as they choose simulation steps during VPS.

Table 18.1: Types of information used by students when choosing simulation steps

Information used	Prototypal scenario	Pattern of usage
Code semantics	'What does this code do and how can we achieve that?'	The semantics of the code are considered and related to the visualization in order to find a simulation step that makes sense.
Program text	'That box looks like what's in the code.'	Superficial resemblance between code and visualization suggests a simulation step.
Unsuccessful GUI operations	'Let's just try everything until we find what doesn't give an error.'	Knowledge of what did not work is used to narrow down the range of possibilities.
Textual feedback	'The system gave a hint there.'	A response from the VPS system suggests a different solution than the one that was tried.
Instructions / Examples	'The example showed that we should do this.'	The current situation is matched to given instructions or a previously seen example that suggests a solution.

Category: Code semantics

The meaning of the program code informs some of the decisions that students make during VPS. A student can attempt to figure out the semantics of the program code, and use his understanding of the program to choose (correctly or not) a simulation step.

The following episode illustrates how John_{36} and Kate_{36} discuss the meaning of code in conceptual terms – speaking of lists, values, and variables – and identify GUI operations that match the intended program behavior.[2]

> *Pair 36 are working on the line* `first = [10, 5, 0]`.
> **John$_{36}$:** So now we have to first create one of those lists. It's, umm... I guess you have to, like, right-click... in there somewhere... there in the empty space. And then "Create in heap"... "list"... yeah. ... And now we put in the values.
> *A bit later, they have initialized the list.*
> **John$_{36}$:** And then... then... mmm...
> **Kate$_{36}$:** We, like, give it the name "first".
> **John$_{36}$:** Yeah. So we need to... create... a variable, so you have to right-click there... that's right.

The second example below illustrates how dialogue between pair members can lead to successful conceptual reasoning. Here, one of the students fails to come up with the correct simulation step until his partner gets him to stop and reflect on what the code really does:

> *Pair 29 is trying to deal with the line* `list = [3, list[0]]`*, but have run into some trouble. A couple of early guesses have failed to yield progress.*
> **Kate$_{29}$:** No-oo... What does that sort of thing do?
> *John$_{29}$ appears to ignore his partner and tries a number of different simulation steps in quick succession – e.g., dragging the literal 3 to the evaluation area and creating new variables – to no avail.*

[2] The aliases of the students are the same as in the previous chapter (see p. 280). Unsubscripted aliases refer to interviewees, subscripted aliases to the paired students whose work we observed.

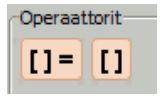


Figure 18.1: List operators in the early UUhistle prototype used by the students. The left-hand operator is used for assigning to a list and the right-hand one for reading a list.

> **John$_{29}$:** No! *Small laugh.*
> **Kate$_{29}$:** *Grunts frustratedly.* What does that, like, mean... three comma list zero?
> **John$_{29}$:** It's like, it now creates... a list with this three in it, and this list zero, the first value.
> **Kate$_{29}$:** Yeah... So...
> **John$_{29}$:** *Brightly:* Should we create a new list here? Could that be the thing? *Clicks on the heap and creates a list object.* Alright!

Category: Program text

Some student reasoning draws on the similarities between the program text, as written, and the visual elements present in the VPS system.

For instance, the line of code `first = [10, 5, 0]` and the assign-to-list operator of the UUhistle prototype (Figure 18.1) look similar.[3] This suggests a simulation step to Pair 17:

> **Kate$_{17}$:** Shouldn't we do it so that we create a "first" and move them into it so it creates a variable? *Points where variables are created.*
> **John$_{17}$:** But, hey, right there we have brackets like that and an equals sign.
> **Kate$_{17}$:** Mmm... *Brings the cursor to the assign-to-list operator.*

Another example of this category is provided by Otto, who, during his interview, mentions trying to match the line of code to what he sees in the visualization.

> *Otto is struggling with the line `if not data_ok(name, age):` and is making no progress despite trying many different simulation steps.*
> **Interviewer$_{2}$:** Could you tell me something about your reasoning, how do you know what to do when you reach a line? For the most part it's gone fairly well... you've known at each step what to do...
> **Otto:** Generally speaking, within each line there are these... *Circles the entire line of code with the mouse.* And then I can also find the things with the same names over here. *Moves the mouse cursor all about the right-hand side of the display.*

Category: Unsuccessful GUI operations

UUhistle informs students when they make a mistake, so each incorrect simulation step allows the student to rule out that option and narrow down the range of candidate steps to consider. Such elimination can be useful whenever a mistake is made. It may also be used as a strategy unto itself. Students comment on this explicitly:

> *Pair 13 are working on a VPS exercise. Kate$_{13}$ hasn't used UUhistle before, so she asks her partner about it.*
> **Kate$_{13}$:** What's the point of this whole thing? It always shows you what's wrong, so you just try all the possibilities?
> **John$_{13}$:** Yup, it's like that... it doesn't have a point.

[3] The list operators in the most recent version look a bit different; see the UUhistle web site.

As they start to work on the first VPS exercises, John$_{20}$ explains his approach to his partner:
John$_{20}$: Usually I just do it so that I first drag... *Small laugh.* I drag these about until it stops complaining. *Embarks on a trial-and-error sequence of missteps and undos.*

Pair 21 have been working on a VPS assignment fairly fluently, but have run into trouble when they are expected to pass parameters into the frame they have just created.
Kate$_{21}$: As long as it doesn't complain, you've got it right.
John$_{21}$: Yeah.
Kate$_{21}$: Usually. Not always, but usually.
After a few tries the pair manage to create one parameter variable and give it a value, but get stuck again.
John$_{21}$: So next it's "test", I suppose? *They try creating a variable `test`, which is not the correct step.* No. A small mistake.
Kate$_{21}$: This is getting pretty random.
John$_{21}$: Should we dump the first one there? *Moves the value of the first parameter variable – which they have managed to initialize already – into the expression evaluation area.* No.
Kate$_{21}$: Let's cover all the options, then it's gonna work....
John$_{21}$: Should I now dump this thing over here? *Fetches the function definition from the Functions panel into the evaluation area.* No.

Category: Textual feedback

The UUhistle prototype that the students used gave simple, fairly generic textual feedback about some missteps (in the vein of: "Incorrect type of execution step.", "You gave the wrong name to a variable.", "You didn't start executing the right function."). In UUhistle v0.2, the user requested this feedback by pressing a button after they made a mistake (see Section 16.4.2). We observed a few cases of students using the feedback button and reasoning about the next step with the help of the explanation they received.

Pair 35 have just created a frame for a function call that takes two parameters, the first of which is called `list`. A variable called `test` will be created on the first line of the function body (later, after the parameters have been passed).
John$_{35}$: Over here, we...
Kate$_{35}$: *Interrupts.* Here we need to create these, the function's... I suppose we're in the function's... I suppose we're there, inside it? *Waves the mouse cursor about in the function body.*
John$_{35}$: Yeah... Over here, we need to create... "test".
Kate$_{35}$: "Test"... *Creates the variable.* No-o... now it gives some error. *Clicks on the feedback button.* Wrong name!
John$_{35}$: Is it "list" then?
Kate$_{35}$: Yeah, it should be that "list". *Creates the variable.*
John$_{35}$: Yeah. And now if you drag into it...

Category: Instructions/Examples

Sometimes, students justified their choice of simulation step by referring to the operational instructions that they had been given concerning how to carry out a task in UUhistle. A similar source of information is the program visualization examples the students had previously seen, which demonstrate how to deal with a particular kind of program.

William here relies on instructions to tell him what to do, and also remarks that the program animations help:

William is just beginning to work on a VPS exercise that starts with a function definition.
William: Well, like, I'd think that it needs to be named first and only then... like... start defining what it is that it contains.
Interviewer$_1$: Okay. And you base this on... what?
William: Well... *Laughs.* I guess on the fact that it says so here in the instructions.[4]
After a bit of discussion about other topics, William refers to the importance of examples.
William: These [UUhistle assignments] are usually easier to do once you've watched the example of how to do it, so you get this, like... um... idea of how you're supposed to do it.

About the categories

The categories listed above map out some types of information that features in students' argumentation as they choose simulation steps. It is important to note here that we have not categorized people. A particular student or pair of students may use different tactics and different kinds of argumentation at different points of even a single VPS exercise. (Indeed, our data suggests that such changes of tactics are very common.) For instance, a tricky simulation step that the students already made a mistake with can be approached quite differently than a familiar step.

Vague spots

Our categorization is based on the arguments and directly observable actions of the students. We looked at what information we could see and hear students using as they make choices, and what information they cite as important. In some of the categories, it is relatively clear how a particular kind of information contributes to decision making. In other cases it is less clear, perhaps especially so in the Instructions/Examples category. Our categorization does not specify *how* the students make use of the instructions and examples. We may speculate about distinct usage scenarios: a student may simply perform steps mentioned in the instructions without a deeper understanding of what they are doing, or they may use the operational instructions to complement their reasoning about code semantics (to find the correct GUI operation that matches what they are trying to 'make the computer do'). However, our data is quite opaque on this matter and merely allows us to vaguely conclude that students do refer to instructions as they make choices.

Our data does not enable us to see inside students' heads. In our recordings, students often chose simulation steps in silence, or simply stated that they "must" or "should" now perform a particular step. In many cases, we could not determine why a step was chosen, at least not without relying heavily on guesswork. Our categorization enumerates some of the things that inform students' decision making during VPS, but the categorization should not be taken as an exhaustive list.

18.2.2 We observed instances of learning

We witnessed various episodes where students learned something as they worked on a VPS exercise. To illustrate, we have selected a few examples, below, that feature different topics and different ways of arriving at learning.

Example: Learning about references by reflecting on the visualization

Pair 6 have been working on these lines of code:

```
second = [3, 1, 3, value]
third = second
second[3] = 100
print second
```

[4]The instructions for the VPS assignment that William is working on say that the student should define the functions in the Functions area when they arrive on one of the `def` lines at the beginning of the program.

```
print third
second = third
```

As they arrive at the last of the lines, Kate_6's attention is drawn to the values of the two variables at the same time, and she notes that they are identical. The reflection that follows helps the pair learn about reference semantics:

> **John_6:** Right... then it's going to do that again...
> **Kate_6:** I got it! It's like... You see, these refer to the same thing now. *Moves the mouse cursor between the references stored in `second` and `third`. Each mouseover causes the modified list to be highlighted.* Look, there... It put the hundred.
> **John_6:** Oh yeah, they do refer [to the same thing].
> **Kate_6:** So, like... So it didn't, like, create a new list after all...
> **John_6:** That's right.
> **Kate_6:** ... instead, when the "second" gets moved to the "third" place, and then you change "second", then it changes the "third".
> **John_6:** You're right.
> **Kate_6:** Pretty cool.

A possibly significant aspect of this example is that the learning occurs not as the second reference to the list is created – the pair had created the variable `third` and copied the value of `second` into it quite effortlessly and without much apparent reflection – but only later as another assignment statement draws the students' attention to the two references. We will return to this topic in the next chapter.

Example: Learning about function calls the hard way

Pair 24 have great trouble with – but eventually succeed in – processing a function call. The following transcript paraphrases their attempt to deal with parameters, which takes several minutes.

> *Pair 24 are processing the line `print weird(list, 1)` which calls a function defined as `def weird(list, index):...` They have successfully evaluated the parameters and created a frame.*
> **Kate_{24}:** *Tries a few entirely incorrect steps in quick succession.* Just throwing things about isn't working.[5]
> *They stop to think for a long time, then again try dragging various visual elements to various places, and creating more elements.*
> **John_{24}:** Could we print that "weird list" and then we'd get in?
> **Kate_{24}:** Yeah, but it's over here... and now we're basically in this section that's orange, where we define that... that... that... *Trails off.*
> *They stop to think again.*
> **Kate_{24}:** Now it has called it down there. Then we've like created a function [call] of it, there, next. What does it want now?
> *Kate_{24} takes the first parameter value from the calling frame and drags it to the evaluation area of the top frame. They then try a few more things. John_{24} keeps promoting the idea of trying to print something, but Kate explains that that can only be done after the function is dealt with and returns something. This brings her to discuss her understanding of the meaning of the code:*
> **Kate_{24}:** [It will] print "weird list" of these two values. *Points at the parameter expressions `list, 1`.* So it does the function. It returns this... *(Points at the return statement within the function.)* and prints it.

Kate_{24} has some idea of how functions work: she wants to produce the return value of the function and has an idea that the values of parameter expressions must go 'into the function' somehow. Neither of the

[5] Kate_{24} repeatedly refers to her occasional attempts to solve the exercise by naïve trial and error as "throwing things about".

students appears to think of the parameters as variables, however, which is the perspective that UUhistle is designed to illustrate and teach about.

The pair continue working on the problem:

> *After trying an impressive array of increasingly desperate incorrect steps, including an attempt to create more operators, Kate$_{24}$ stops to think again.*
> **Kate$_{24}$:** Hmh! Makes no sense at all. *Mumbles something inaudible.* "list"... and "index"... And then we have "list"... and one...
> *Several more unsuccessful attempts at "throwing things about" follow.*
> **Kate$_{24}$:** Maybe we should create one of these? *Creates a variable called* `index`. No. *Creates a variable called* `list`. Oh wow!
> **John$_{24}$:** Huh?
> **Kate$_{24}$:** I created "list" there because I thought that since we have these over here. *Mouses at the function signature where the parameters are named.* So I thought maybe we should first create them here. Now we should move this here... like so. *Drags the list reference from the calling frame to the new variable.*
> *They process the second parameter without any trouble.*
> **Kate$_{24}$:** That's it! Now we have... in the function we have defined... these. *Mouses over the new variables that now have values.*
> **John$_{24}$:** Mmm.
> **Kate$_{24}$:** From there. *Mouses towards the calling frame.* And it sure took a while again. These are always like this. You throw things about for a while and then... it happens. And then maybe with luck you figure out, like this, afterwards, what has happened.

The passage shows that Pair 24 did not originally know how to pass parameters in UUhistle. They also appear to have had a limited understanding of parameter passing in general. The idea that there is a variable (or any kind of storage) corresponding to each parameter is conspicuous by its absence from their reasoning. Their (or Kate$_{24}$'s at least) understanding seems to be solidified by the VPS exercise: Kate$_{24}$'s summary at the end suggests that she understands the purpose of what they eventually accomplished, even though she had little idea of what to try for earlier. John$_{24}$, whose understanding of the topic appears to have been more limited, perhaps also learned something about how return values work in nested expressions.

Later in the same VPS exercise, there is another function call, which the pair deal with fluently.[6]

Example: Learning about recursion almost as a matter of course

Two of our interviewees, Sue and Elizabeth, did a VPS exercise during their interviews in which they had to simulate the behavior of a recursive factorial function. Neither had seen a recursive program before. Both learned to understand the program with the help of UUhistle during the interview.

Asked to reflect on her learning, Sue described how UUhistle helped her get an idea of how some of the calls stay "in queue" waiting for the others to complete as the factorial is being calculated. Elizabeth explained how UUhistle helped her understand that multiple frames can correspond to the same function definition and that the recursion retracts in the call stack "like a snowball". (Longer quotes from the two students appear in Section 17.4.6 in the previous chapter.)

A potentially significant aspect of the two cases we observed is that not only did both students learn about recursion, but neither appeared to find the program troublesome. This is despite the fact that recursion is considered by many in the computing education community to be one of the most challenging topics in introductory programming (see references in Section 3.4). Both students very quickly found the correct simulation steps needed for the recursive call, although the process did provoke some thoughts. Here is Elizabeth:

> *Elizabeth is thinking aloud as she is simulating the call* `return factorial(n-1) * n`. *She has just evaluated* `n-1` *to yield 4.*

[6] The pair also go on to produce a perfect answer to the post-test question on (nested) function calls in our experimental setup (Chapter 19); they had had only a partially correct answer for the same question in the pretest.

Elizabeth: And... So the function is called again. I suppose that means that it sorta... I wonder if I can create, here, still another... *(Clicks the heap to bring up the "new frame" option but does not click it.)* frame, I guess I can? Or is it so that the same [frame] can... In the same, like, umm... If the [sub]program has been called *(Circles the mouse cursor around the code of the function.)* is it so that I can put it there again in the same [frame]? So that instead I give this "n" a value within the same [frame] thingy that I've been working with *(Motions with the mouse from the parameter value 4 in the current top frame towards the variable `n` in the same frame, which contains 5.)* or do I have to create a new frame? *(Clicks the context menu to produce a new frame. No error: this is the correct step.)*
Interviewer$_1$: Feel free to try.
Elizabeth: *Laughs.* Well, apparently I can [create a new frame]. *Proceeds to create another variable `n` in the new top frame and brings the value 4 there.*

In this quote, Elizabeth considers two ways of dealing with the recursive program: the correct solution that uses multiple frames and an alternative solution that relies on a single frame and is not viable here. She decides to go along with her first hunch of creating another frame and finds out that it works. (The alternative 'looping model of recursion' is reported in the literature as a common way of misunderstanding recursion; see Appendix A.)

Our observations are intriguing in that they concretely show that it is possible to learn to understand recursive programs by applying one's understanding of a notional machine. In UUhistle's notional machine, there is nothing special about recursive function calls compared to other nested calls. Elizabeth and Sue learned to deal with self-calling functions quite naturally and painlessly, building on their existing knowledge of function calls.

The above quote from Elizabeth also illustrates how the VPS system can serve as a platform for the interactive exploration of ideas, supported by automatic feedback: had Elizabeth chosen instead to try out the 'looping model', she would have received an error message and would then presumably have tried the correct alternative.

We are not in a position to generalize these findings to any other students or to claim that VPS or UUhistle makes learning to understand recursive programs easy. Further study is needed to establish the extent to which these interviews are representative of CS1 students in general.

18.2.3 We saw a few pedagogically interesting patterns of student behavior

A few patterns of student behavior during VPS work stood out as potentially significant from a learning point of view. We comment on these below.

Revisiting one's goals... or failing to

One theme that emerged from the analysis was the way some students succeeded in overcoming problems by returning to reflect on what they were trying to accomplish through the GUI. Other students failed to revisit their goals and struggled enormously.

Analytically, we may separate what the VPS user does as they choose what to do into two parts: deciding what one wishes to happen next, and choosing a GUI operation that (one hopes) achieves this intended goal. Students do not necessarily draw this distinction, however. Even if they do, they may fail to consider that the root cause of any mistake they make can lie within either part. Let us contrast two examples.

Pair 40 initially make the wrong assumption about what a line of code does, thinking it modifies an existing list. This gives them a headache, which is resolved as soon as they consider an alternative meaning for the code (it creates a new list) and find a GUI operation to match the new interpretation.

Pair 40 are discussing the line `list = [3, list[0]]`.
John$_{40}$: Right, so it kinda shifts the list one step forward and adds the number three in front, doesn't it?
Kate$_{40}$: Mmm... let's try.

John$_{40}$: At least I'd imagine it does.
Kate$_{40}$: Yeah.
They try creating a new reference to the existing list in the evaluation area.
Kate$_{40}$: No.
John$_{40}$: No good.
Kate$_{40}$: Then...
John$_{40}$: I think this just adds the three to the list..
Kate$_{40}$: Okay. Yeah.
John$_{40}$: Let's try one more thing. *Drags a reference from the variable `list` to the evaluation area, to no avail. Sighs.*
They pause to think in silence for a lengthy spell and try a couple more incorrect options for assigning to the list.
John$_{40}$: *Brings the assign-to-list operator to the evaluation area.* This assigns something to a certain index. *Again brings the reference from `list` to the evaluation area.* It's going wrong.
Kate$_{40}$: I'd think that's the right one, but...
John$_{40}$: Me too. That we should put it here first and... assign with these *(Motions at the operators.)* to an index.
Kate$_{40}$: Mmhmm... Hmm.
John$_{40}$: What if we assign first – is this possible? – the three. *Brings the literal 3 from the heap to the evaluation area.* No. No. Nono. I'd imagine that we could dump the three there, then the list after it.
Brief pause.
John$_{40}$: I would understand if we had to create... a totally new list.
Kate$_{40}$: Right.
John$_{40}$: But I bet that's wrong too. Let's try it. *Creates a list.* It wasn't!

Let us now consider the case of Otto, who sticks with great persistence to his idea of what happens next. Otto is working on the following code:

```
if not check_info(name, age):
    return False
print 'First name:', name
print 'Age (years):', age
return True
```

Otto has just successfully returned `True` from the function `check_info` and applied the `not` operator to produce `False` as the result of the conditional expression. (The next correct step is to indicate which line to jump to, in this case the first print statement.)
Interviewer$_2$: Okay, so what now?
Otto: Well, now we get to return this "False" to the other [sub]program... *Drags the value into the calling frame to return it, but UUhistle signals an error. The interviewer starts to ask about Otto's reasoning, but Otto fails to answer and decides to pursue different ways of returning a value. He clicks on the next line – `return False` – in an attempt to move program control there. Seeing that Otto has great trouble advancing, the interviewer eventually hints that the line wasn't the correct one. Otto eventually manages to click the right line but appears puzzled:*
Otto: [So] that's where we went.
Interviewer$_2$: Does it make sense to you that we went there, or...?
Otto: A pretty weird thing.
Interviewer$_2$: If you had to give your best guess why we went to that line number seven [the first `print`], what would it be?
Otto: It returned "False" and that's why we continue from here.
A while later, another function call brings Otto back to the same situation, only this time the conditional has evaluated to `True`.

Otto: "True"... And then we go... to return "True". *Clicks on the line* `return True` *to move control there.* No, we didn't go there. *Clicks on the first print statement.* We went here last time, no, not there, either.
The interviewer tries to initiate a discussion about the meaning of the program but Otto has little to say beyond the fact that the program checks that some numbers are equal to or greater than zero. Discussion returns to the present problem.
Interviewer$_2$: So, now we are on line five and we have that "True" there?
Otto: Yeah, so... But that we can probably... take here right away. *Returns 'True' from the top frame into the calling frame.* No, we can't.
Otto continues with various attempts to make the function return 'True'. He tries dragging the boolean literal from the heap, transferring control to the calling function, and repeating earlier attempts. Pointing at the (inactive) line `return True`*, he reiterates:* The program wants to return "True" here. *Otto does not manage to progress past this point during the interview. After the interview, the interviewer explains to him how the program works.*

During the interview, Otto decides, twice, that the next step is to return the boolean value that the conditional expression evaluated to. We do not know the precise reasons for this (he seemed to struggle with `if` statements and may have a misconception concerning them and/or the `not` keyword; he may also have been misled by UUhistle's way of dealing with these topics). Whatever the reasons were, he does not appear to seriously question his conclusion at any point. Laboring under the wrong impression, Otto wastes time looking for an answer to the wrong question of which GUI operation allows him to return the value that is currently in the expression evaluation area.

From trial and error to reflection

We observed cases where eliminating incorrect answers through trial and error did not appear to lead to learning even after the correct step was found (often after considerable trouble). The students would just move on to the next step without ever thinking about the meaning of the current step.

We also observed some cases where learning did appear to take place after trial and error. Kate$_{24}$ sums this up on page 309 above: you try all kinds of things without much thinking and only after you find out the correct answer do you get what the point was. Kate$_{24}$ does look for meaning in the visualization but starts "throwing things about" when she runs into trouble, then reflects on what she has accomplished. Her overall approach appears to be deeper than what one might think if one were to only observe the trial-and-error tactics she resorts to when she feels she needs them.

Sue, one of our interviewees, also discussed a way of using trial and error as part of a deeper learning strategy. Sue says that when she worked on the VPS assignments of the course, she sometimes used "a trial-and-error tactic" to find the correct answers and get the points, then went back to the VPS task to make sure she understood each of the steps. This may even involve redoing the exercise several times. Here is a selection of quotes from Sue:

On doing VPS in the past:
Sue: Many times I've done so that I didn't know what to do, so I clicked many times on different options until it didn't complain, and then I went forward from there.
After she completes a VPS exercise during the interview:
Sue: Now I get the... I got this logic, but I didn't internalize it ... I'd need to do this so many times that I get every click right, then I'd maybe understand the logic.
Again on how she works on the assignments:
Sue: When I've gotten stuck, I haven't thought about them too carefully *(Laughs.)*, I've tried to do something and as soon as it doesn't complain I've known it's okay. And then I try to think about why it's okay.
Interviewer$_1$: Mmm.
Sue: So I don't, in a way, try to solve it beforehand, only when I know the result I try to solve it in the sense that I think about why it went like that. It's generally my way of learning that I first see "how is this done", and then I start thinking about how... why it's done like that.

Not so that I first look and start thinking right away about why it's done. ... It is a bit... a bit tricky, but luckily you can do them many times in UUhistle. So it isn't, like, after you turn it in once and get full points after you've struggled with it for an hour[7], you don't get to do... So that you get to do it again so many times that you can then... like... basically, you redo it a couple of times after you've done it once. So, after that it starts getting easier when you... when you know what to do, so you can start thinking about why you're doing it, but in the beginning it's more like... it easily becomes, like... you don't necessarily, like, think why this goes there, because you have no clue.
Interviewer$_1$: Mmm.
Sue: About what you're trying to do. But then when you get something to go somewhere... then you get... it gives you this clarity, things start rolling, so to speak.

The author of the thesis noted in Section 14.2 that a program animation is, in a way, a worked-out example solution for a VPS problem. Seen from this perspective, we could describe Sue's strategy as mechanically turning problems into worked-out examples, which she then studies in order to understand how to really solve the problem.

An unusual conclusion: UUhistle 'lied'

At the beginning of Section 18.2.2, Pair 6 saw that two references pointed to the same list and used this to draw valid conclusions about program semantics. Pair 14 saw the same evidence but came to a very different conclusion.

```
second = [3, 1, 3, value]
third = second
second[3] = 100
print second
print third
second = third
```

Pair 14 have successfully simulated the program's execution until the last line quoted above. They are just noticing that there is only a single, modified list in the heap which both variables refer to:
John$_{14}$: How can it first, over there... that this "third" equals "second"? It goes all wrong now! Now that it's been defined there, this "third". *Points at the code* `third = second`.
Kate$_{14}$: Yeah.
John$_{14}$: So it's not supposed to... when we changed that "second" afterward... [not supposed to] make it so. *Points at the modified list in the heap.* But it didn't yet...
Kate$_{14}$: It's not?
John$_{14}$: But it didn't complain yet?
Kate$_{14}$: But... doesn't it precisely because we define it first ... and then we change "second".
John$_{14}$: It's not supposed to, because...
Kate$_{14}$: *Interrupts:* Right, [we changed] the original "second", right...
John$_{14}$: Like, if you define that "my name" is "your name", and then we change your name afterwards, it shouldn't change my name.
Kate$_{14}$: Right, yeah, right.
John$_{14}$: But it didn't whine about it in any way so I suppose this *(Moves the mouse cursor around the references stored in the variables.)* is a feature in this [software].
Kate$_{14}$: But now we have again: "second" equals "third".
They carry out the assignment correctly.
John$_{14}$: Doesn't seem to be going right.

[7] According to our logs, students extremely rarely spent an hour or longer on a VPS exercise. It is very likely that Sue is exaggerating here.

The pair fail to accommodate reference semantics within their understanding of assignment and variables, and resist the alternative view of program behavior suggested by UUhistle. They decide that references are merely a misleading "feature" of UUhistle. In their later work during this session they show no sign of reconsidering this view.[8]

This was the only case we observed in which the students explicitly challenged the fidelity of the visualization.

Ignoring aspects of the VPS environment

Let us now consider some things that students did *not* do and or pay attention to.

Again, the following analysis is based on dialogue and students' use of the mouse. We did not, for instance, use eye-tracking equipment, nor did we have video of students' bodily gestures. Nevertheless, we can comment on a few elements within the VPS system, which, our data suggests, some students entirely ignored.

Many of the students we observed had already ignored some available sources of information before they started on the VPS task proper.

The first VPS assignment was prefaced with short instructions, which were visible onscreen before the simulation task started. The large majority of the students clearly did not read these at all or only granted them the briefest of browses.

Numerous student pairs also ignored the program animation provided as an introductory example of the topics covered (see the next chapter and in Appendix C). The example of lists and references appeared in UUhistle's assignment selection menu right before the two VPS assignments. However, nearly half of the 41 pairs did not start by watching the example, instead jumping directly to the VPS task (usually without reading the instructions for that task, either). From our data it is clear that failing to read the instructions was invariably associated with significant puzzlement and trial-and-error tactics regarding list operations right at the start of the first VPS task. The students who had skipped the example and the instructions had little idea of what steps are involved in carrying out a list operation in UUhistle and in what order (e.g., to produce a new list, you had to first create a list in the heap, to form a reference to it in the evaluation area, and finally to initialize it). Only after flailing about, usually clearly frustrated, did the pairs eventually figure out how to move forward. We must be careful as we consider the causalities of the situation. It is possible that another factor (e.g., a certain attitude or approach to learning) led some of the students both to skip the example and to struggle with the list operations. Nevertheless, it seems to us a very reasonable conjecture that the failures to look at the example and the instructions contributed significantly towards the students' troubles.

After they completed the second VPS assignment, students often checked UUhistle's assignment menu again to see if there was anything more they were required to do. At this point, many of the pairs who had not initially watched the example animation did go back to watch the example. Viewing the example afterwards almost invariably led to dialogue such as this:

> **Kate**$_{33}$**:** This is one of those where you have to just, like, click your way through it.
> **John**$_{33}$**:** Ow! If we had only watched this in the beginning, this one!
> **Kate**$_{33}$**:** Uh-huh. *Laughs without humor.*

During the VPS exercise itself, students' attention was primarily on the program code and the graphical elements they manipulated. The majority of pairs did not explicitly refer to program output at all. Even fewer pairs appeared to reflect thoughtfully on what was printed out. None of the students ever looked at UUhistle's menus at any point for any purpose except to change assignments.

The vast majority of the pairs never used the feedback button to request textual feedback on their solution. The example on page 306 is one of only a handful of individual episodes where someone did ask for feedback.

[8] From a pedagogical point of view, this is not exactly a happy outcome. However, we may still speculate (and hope) that this experience may have been a memorable one for the pair, and they perhaps came to recall it at some later stage of the programming course, such as when references created a bug in their own program.

18.2.4 We catalogued trouble spots and student mistakes

Within the two VPS exercises that the student pairs worked on, a few sections stood out as particularly troublesome for the students. We comment on these below and discuss some of the specific mistakes that students made.

We did not analyze in detail every single mistake the students made, nor can we present an exhaustive list of all the ones we did spot. We focus on recurring mistakes related to the topics that the students had most trouble with. The presentation below excludes trivial slips of the mouse and the 'flailings' of students selecting multiple steps in quick succession in a trial-and-error process.

The two main problem areas we identified were list operations and parameter passing.

Problems with list operations

At the beginning of the first VPS exercise of the session, students had to create a list of integers in the heap, initialize it, and then use list operators (Figure 18.1) to access the list:

```
first = [10, 5, 0]
first[1] = -5
value = first[2]
```

Many pairs made mistakes with these steps. They would attempt to use the wrong operator (read a list instead of writing or vice versa) or try to use the operators to create lists (which is done in UUhistle by clicking on the heap).

Lists were a topic that had just been introduced in CS1–Imp–Pyth. None of the students had previously seen UUhistle's visualizations of lists or references. From the dialogue, it seemed that most students had roughly the right idea about what these lines of code did; however, they were not sure what simulation steps were involved and how to perform them in UUhistle.

Confusion about the steps involved and the meaning of the visualization was extremely common among those pairs who had not watched the program animation on lists before they started working on the VPS task.

A point on usability can be made here. Lists and list operations (and references to lists) were the only novel elements in the visualizations used during the session we observed. The students had previously encountered variables, assignment, function calls, etc., in UUhistle. This, together with the fact that many students did not watch the example animation, goes some way towards explaining why the students had trouble with finding the right GUI operations for basic list operations. Such trouble was conspicuously absent elsewhere in our data, however. That is, apart from the list operations at the beginning of the first VPS exercise, we observed extremely few situations in which the students expressed a conceptually correct idea of what the next simulation step should be but nevertheless had significant trouble with finding the corresponding GUI operation. This suggests that many students had learned to use UUhistle's GUI to carry out the execution steps they wished to be carried out.

Beyond the first few lines of the first VPS exercise, there were two more trouble spots concerning list operations. The line `list = [3, list[0]]`, which appears in the second VPS exercise, was a source of puzzlement for many pairs. Many initially failed to recognize that it creates a list. Some thought that instead it (only) performs an operation on the existing list, and tried to use the list operators to achieve this aim. No-one who explicitly worked out that the command does in fact create a new list had trouble creating and initializing the list (which they had already learned to do before getting to this line). The use of brackets within a function call in `quaint(['sheep', 'chamois', 'sheep'], 'grandma')` also puzzled some students. Students' difficulties with these list creation commands suggest that their knowledge of list semantics was still fragile and undergeneralized, and they had probably not experienced much variation in the contexts where lists can be created or in the kinds of expressions that can be used to initialize lists.

Problems with parameters

Parameter passing, featured in the second exercise, was by far the most troublesome part of the VPS work that the student pairs did. A clear majority of the pairs had some kind of difficulty with it.

Parameter passing was the only topic about which any of the pairs asked for help from the assistant supervising the session. It was also the only stage at which any of the pairs got so badly stuck that they decided to give up entirely on solving the VPS assignment. (Only a few pairs did either of these things, but all the cases involved parameter passing.)

Students' difficulties usually started only after the pair had evaluated the parameter expressions and created a frame. Many pairs ran into trouble immediately at this point. Every step of the parameter-passing process proved troublesome for at least some of the pairs, with students failing to pass parameters at all, failing to create variables to store the parameter values, not understanding which variables to create, and/or failing to realize which values the parameter variables are supposed to get.

A list of the main parameter-passing difficulties we observed is shown in Table 18.2. The relevant code fragments that the table refers to appear below.

```
def weird(list, index):
    test = list
    # ...

def quaint(list, element):
    # ...

list = ['cobra', 'python', 'mamba']
print weird(list, 1)
second = quaint(['sheep', 'chamois', 'sheep'], 'grandma')
# ...
```

Problems with other topics

Two further trouble spots stood out from the data.

The first VPS assignment featured the line `second = third`. Some pairs tried to execute this line by assigning the value of `second` to `third`, instead of the other way around. The reasons are not clear from our data. Possibly, some of the students harbored the misconception that assignment works in the other direction or that the order does not matter (at least in the case of a statement of the form `var = otherVar`). It is also possible that the students were bemused by the fact that at the time this line is executed, the two variables already have identical values, so the assignment does not actually alter the value of either one. This was the only statement in the two VPS assignments that copies the value of an existing variable into another existing variable, so we do not have further data on this matter. The students did not have similar trouble with other assignment statements.

Another tricky line was `return test[1]` in the second VPS assignment: roughly one pair in four appeared to read this as `return test` and returned a list reference rather than one of the list's elements. The mistake was usually noticed and fixed very quickly by the students, and appears to be more of a misread than a deeper misconception. We may speculate that the mistake may have been rooted in superficial code-reading strategies, unfamiliarity with lists, and/or the way parenthesized expressions can be treated as optional in many natural language texts. Our data does not allow us to confirm these speculations, however.

What was not a problem?

Although we did not perform a formal quantitative analysis, we can note that the clear majority of students' mistakes involved the trouble spots listed above. Less problematic aspects included: forming expressions bit-by-bit in the evaluation area (including function calls), creating variables, most assignment statements (including reference assignment), declaring functions, returning values, and creating new frames for function calls.

The distribution of mistakes is, of course, affected by the specific assignments used, the students' existing knowledge of programming, and prior VPSing experience.

Table 18.2: Students' difficulties while simulating parameter passing. See the previous page or Appendix C for the code.

Difficulty	Symptoms	Notes
Failing to pass parameters at all.	Students ignore parameters and attempt to proceed past the phase either by moving control to the function body or simply by starting to execute the body.	This behavior was very common in our data set. The root causes of this rather opaque behavior may lie in forgetfulness, failure to understand parameter passing, or failure to realize that UUhistle requires students to pass parameters explicitly.
Failing to make use of variables.	Students do not create variables and instead try to drag the parameter values into other parts of the new frame, usually the expression evaluation area.	May be caused by a failure to understand the role of formal parameters in function definitions, a failure to think of parameter variables as regular local variables, or a failure to realize that UUhistle requires each parameter to be explicitly stored in a variable.
Name-linked parameter passing, variant 1	Students only create a `list` parameter variable into the new frame for `weird`. They give it a value directly from the variable with the same name in the calling frame, and then try to advance inside the function body without creating `index` at all.	May be indicative of a misconception that variables in the calling code and the called functions are 'linked by name'. (This misconception is reported in the literature; see Appendix A.) The way students ignored `index` suggests that they may only have looked at the function call `weird(list, 1)` and created only `list` because 'it's the variable that is passed to the function'.
Name-linked parameter passing, variant 2	Students give the `list` parameter of `weird` a value from the calling frame's `list` variable. They correctly create the variable `index` but are puzzled as to where it might get its value from.	May also be suggestive of the same misconception as the problem above. In this variant, however, the students clearly created the parameter variables on the basis of the function definition rather than the function call.
Variable-to-variable parameter passing	Students give the `list` parameter of `quaint` a value from the calling frame's `list` variable.	This variant of the two problems listed above may be indicative of a misconception (reported in the literature; see Appendix A) in which parameter-passing is seen as linking variables of the caller and callee.
Avoiding duplicate variables	Students do not pass the first parameter `list` of the function `weird`. They only pass the second parameter `index`. Once they find out through the system that they have to create `list` in the new frame as well, they sometimes wonder aloud about the need to do so since "we already have one".	Suggests a limited understanding of parameters and scope.
Calling a function again instead of processing the original invocation.	Instead of passing parameters, students form another call to the same function in the evaluation area of the new frame they just created.	The function signature is highlighted by UUhistle when the student is expected to pass parameters. Students may react to this by fetching the graphical element that most closely resembles the active line of code. Students may also be confused by how UUhistle first highlights the signature when the function is being declared and then again when it is being called. This step was also sometimes used as part of a naïve trial-and-error strategy.
Problems with nested syntax.	Students try to execute the `print` statement in `print weird(list, 1)` when they are supposed to pass parameters.	May be caused by a misconception concerning nested calls or by a failure to realize that the student is expected to execute each step of the `weird` call manually (instead of the call working as a black box). This was also used by some students as a desperate move when nothing else seemed to work.
'Overeager' variable creation.	Students create the first parameter variable, then create the second immediately, without first passing a value to the first variable.	In a different notional machine for Python, this could be correct. In that of UUhistle v0.2, it was not. In the most recent version of UUhistle (Chapter 13) this is a non-issue, because variable creation and the assignment of an initial value form a single atomic operation.

18.3 We have some quantitative results on students' strategies

We now present briefly some quantitative results that complement our qualitative analysis of the observations and interviews. These results are based on the end-of-course feedback survey of the CS1 course, which featured a number of questions concerning UUhistle. Of most relevance to this chapter is a question that asked students to assess how they did VPS: "Which of the following best describe the way you did the assignments in UUhistle (if multiple apply, choose several)?". The students were given nine options to choose from (partially inspired by early impressions of the interviews conducted earlier). Table 18.3 presents the results.

The survey responses suggest that reflecting on the behavior of the program, naïve trial and error, and following instructions were all common approaches to choosing simulation steps. Most students appear to have at least tried to reflect on the meaning of simulation steps either before or after they chose them (73% of the respondents chose at least one of options 1, 3, or 8).

We will have more to say about the feedback survey in Chapter 20.

Table 18.3: Ways of doing VPS, as reported by students in an end-of-course feedback questionnaire. Students were encouraged to choose all the options that apply. The percentages are of the 324 students who picked at least one of the nine options.

#	Option	Chosen by
1	"I first tried to figure out what next happens in the program, and only then tried it to see if I got it right."	143 (44%)
2	"I just tried everything without thinking much, just to get the program to advance."	92 (28%)
3	"I advanced by trying all kinds of things. Still, I tried to think about why the correct step was the right one after I found it."	144 (44%)
4	"I followed the instructions that came with the assignment."	140 (43%)
5	"I followed the instructions I had seen in the introductory videos of UUhistle."	77 (24%)
6	"I did what a friend advised me to do, even though I didn't quite know why."	7 (2%)
7	"I did what a TA advised me to do, even though I didn't quite know why."	8 (2%)
8	"I collected the points for an assignment without thinking about its content, then returned to reflect on why it worked as it did."	18 (6%)
9	"I used some other strategy to work on the assignments."	14 (4%)

18.4 The results suggest improvements to UUhistle and to its use in teaching

The purpose of this study was to explore the novel way of studying that is visual program simulation, and to provoke thoughts and hypotheses of how it works in practice. The results we have presented paint a picture of what goes on during VPS sessions. They demonstrate that meaningful learning about challenging topics can occur – but does not always occur – during VPS and provide some insights into what makes for a successful VPS experience. Below, we consider some of the possible implications of our results for learning, pedagogy, and VPS system design.

18.4.1 Teaching should facilitate reasoning about program semantics during VPS

We can start our discussion of the pedagogical implications of our results by considering the five kinds of information listed in Table 18.1. Taking the purpose of VPS – enhancing the learning of programming – as our point of view, the first sort of information – the conceptual content of the visualization – stands out as key. The more time we get students to spend on reasoning about VPS in terms of program semantics and the notional machine, the more effective their learning of programming is likely to be.

From this perspective, all the other kinds of information are of secondary importance. Making use of them during VPS is desirable only to the extent that they contribute towards reflection on program semantics.

Following examples or instructions is a pedagogically acceptable way of solving VPS exercises when students understand, on a conceptual level, what they do. The need to find the correct simulation steps motivates (some) students to pay attention to the examples and instructions and to learn to follow them. To the extent that this also brings about learning of the concepts present in the examples and instructions, the learning is meaningful. However, if students choose simulation steps by 'mindlessly' following instructions or mimicking program animations, without ever reflecting on meanings, then the process is not likely to produce major insights. Similarly, the usefulness of textual feedback for learning programming (as opposed to just passing the assignment or learning to pass such assignments) is dependent on how well it helps the student to (immediately or eventually) reach a conceptual understanding of program execution (as opposed to only helping him learn where to click).

Two of the categories suggest the existence of what can be termed surface approaches to choosing simulation steps. In the second category, the choice of simulation step is affected by surface similarities between code and visualization. The third category shows that students narrow down the range of possibilities by eliminating erroneous options that they have tried. Neither of these forms of reasoning is ideal from a pedagogical point of view, as in them cognitive effort is not primarily directed at the intended content of learning but at secondary issues.[9]

A pedagogical view of the categorization summarized in Table 18.1 therefore suggests that teaching should seek ways to encourage reasoning about program semantics when choosing simulation steps, and discourage guesswork, the mechanical following of instructions or examples, and trial-and-error strategies. That is not quite the whole story, however.

The most important: a deep overall approach to VPS

It is clear from our data, and an intuitively appealing notion, that failure to consider the conceptual content of the visualization while choosing simulation steps was often associated with a failure to learn about that content. Nevertheless, we also saw evidence of the use of what were ostensibly surface-level tactics for choosing steps used in the service of a deeper goal of learning about programming. The most striking example of this was the case of Sue (p. 312), who resorted to trial and error as a step-finding tactic when in trouble, but returned to reflect on the visualization after she found the correct step.

We may speculate that Sue's time might have been used more productively if she had been helped to relate the visualization to program semantics at each step. This would have required better support from the system and/or a teacher than what Sue had, so that she would not have felt she needed to resort to trial and error because she "had no clue" as to what to do. However, even if Sue's use of time and effort was not optimal, the learning outcome was excellent as her approach to learning through VPS was very deep. She also enjoyed using UUhistle.

We conclude that while using trial-and-error tactics often indicates a surface approach, this is not always the case. Ultimately, it matters less how the student manages to find the correct simulation step. The most important thing is whether the simulation activity brings them to reflect on that content at some point of the process. The occasional strategic use of trial-and-error tactics or other surface approaches to choosing a simulation step is not a problem, if the student's overall approach to the VPS activity is deep

[9] Booth (1992) reports that novice programmers sometimes approach program writing without regard to the problem to be solved: they opportunistically either try to use constructs from their existing repertoire in the hope they will do something useful, or try to tinker with an existing program that has superficial similarities to the problem at hand. We leave to the interested reader the consideration of the parallels between our results, the approaches to program writing identified by Booth, and approaches to problem solving in general.

and the student searches for meaning in the correct answer. That said, even if the student has adopted a reflective approach to VPS, trial-and-error tactics are unlikely to be an optimally effective use of the student's resources.

Pedagogical challenges

A deep learning approach is predicated on a rich understanding of what learning through VPS can be. Together with the conclusion from the previous chapter, the results in this chapter support the view that in order for VPS to be successful, students need to develop a powerful way of understanding VPS so that they realize its potential for helping them learn programming concepts. This is arguably the most crucial pedagogical challenge for teachers who wish to make use of VPS in their teaching.

Another pedagogical challenge is to encourage and help students to use their time as productively as possible when selecting VPS steps. Ideally, students would use as much of their cognitive effort as possible on program semantics and the notional machine that serves to define those semantics, rather than spend it mechanically trying to cover all the options or looking for superficial similarities between code and visualization.[10]

The two challenges are interdependent. A student who experiences VPS in a rich way is more likely to consider program semantics when picking simulation steps (and more likely to harness any other information source they use to the service of the overall goal of learning programming). Conversely, a student who finds programming concepts useful for picking VPS steps is likelier to develop the view that VPS is useful for learning about such concepts.

Encouraging reasoning about program semantics, in practice

The previous chapter has already outlined some pedagogical strategies for enriching students' understandings of VPS by highlighting the conceptual content present in the visualization. We can now extend this advice on the basis of the results we have presented in this chapter. Below, we give examples of student–teacher and student–system interactions that could encourage and facilitate reasoning about program semantics during VPS sessions.

Our categorization provides teachers and VPS system developers with an idea of the kinds of information students use when making decisions. Teachers and systems can try to identify these categories in students' dialogue and actions. Identifying how a particular student (or pair of students) reasons can then serve as a foundation for pedagogical guidance. Teachers should, when possible, engage students in dialogue about how they reason as they choose VPS steps. The teacher may then try to lead the students to reflect on the meaning of the program and how it relates to the visualization, and to help students see that understanding the program and the visualization leads to the selection of the correct simulation step. VPS systems could also aim for something similar by inviting or even requiring students to explore texts that explain the behavior of the program and the meaning of the visual components. Reflective dialogue on the visualization might also be encouraged by having the students work on VPS exercises in pairs.

Both teachers and VPS systems should strive to underline the difference between knowing conceptually what the next step in the program's execution is and knowing what the matching GUI operation is. Students should be encouraged to reflect on the former, and the VPS system should be designed so that if the simulation step is known, the GUI operation will be found easily and naturally. An alternative form of VPS (compared to UUhistle) might even seek to eliminate the distinction entirely. The user would specify what to do not by manipulating graphics but by choosing at each step a textual description of what happens next in the notional machine, and the VPS system would then visualize the consequences. Such an approach would have drawbacks, too, however (cf. Chapter 15).

Students can get into a lot of trouble by failing to question their assumptions about what the given program does (see Section 18.2.3). Students may think they have already solved the problem of figuring

[10]We conjecture that following a given VPS procedure or mimicking an example animation, even without quite understanding what one is doing, can sometimes be a useful part of a learning process. Practicing the procedure of program tracing may eventually contribute towards a conceptual understanding of what one first learns to manipulate in a concrete way (cf., e.g., Sfard, 1991). Nevertheless, a search for meaning should be a part of this process even during the concrete, procedural stage.

out what the program does or may take the solution for granted. When a student experiences difficulties during VPS, the teacher or the VPS system may ask the student to specify what it is they are trying to accomplish next. If the student is not sure, they should be guided to reflect on program semantics and what they have learned about the notional machine. If they are trying to do the right thing but failing to find the correct GUI operation, they can be advised about the GUI. If they are trying to do the wrong thing, they should be invited to challenge their assumptions about program semantics.

Teachers and VPS systems should be open to the fact that students sometimes do not know what they have done even after they have found the correct step. Systems could be designed so that they invite and help the student to reflect also on the simulation steps that have already been completed.

Teachers should be alert for ineffective VPSing strategies. Superficially matching program text to visual elements should be discouraged, as should trial and error. VPS systems could also try to heuristically detect such behaviors and offer recommendations to students on the spot. Systems could also be designed to make naïve trial and error even more difficult and unrewarding than it is in UUhistle, by increasing the number of possible GUI operations, for instance.

Instructions and example animations should be designed so that when students use them to figure out what to do during VPS, they are encouraged to reflect about programming content. Assignment descriptions and other instructions should be phrased in programming terms. Where operational instructions need to be given about what to drag or click, they should be linked to the conceptual content of the visualization.[11] Example animations should be annotated with texts that explain the meaning of the animation. Similarly, textual feedback from the system should be conceptual so that it helps the student to actively figure out the underlying problem rather than suggesting a quick fix. For instance, UUhistle's hint of "You created a variable with a wrong name." (p. 306 above) may be an example of poor feedback that does not address any deeper issues behind the student's misstep but which the student may nevertheless use to find the correct step without ever figuring out what the real problem was.

18.4.2 Teachers and VPS systems should be alert to the reasons for specific mistakes

Awareness of the kinds of mistakes students make during VPS can lead to better pedagogy. Some mistakes may be indicative of significant programming misconceptions. Other mistakes may be caused simply by lack of familiarity with the VPS system and ignorance of given instructions and examples.

Relating mistakes to misconceptions

It was suggested in the previous chapter that highlighting students' misconceptions during VPS can be one way of demonstrating to students how VPS can be helpful for learning programming. Moreover, VPS can serve the teacher in uncovering misconceptions and addressing them.

The results in this chapter go some way towards showing that VPS has potential as a pedagogical analysis tool. As they engage in VPS, students expose their understandings – and misunderstandings – of programs and the notional machine. A teacher, researcher, or VPS system may analyze these understandings for the purpose of helping that student or in order to gain insights into the learning of programming more generally.

Teachers should get students to comment on their own mistakes and encourage them to look for the reasons behind those mistakes. Teachers themselves, as they observe students working on VPS, should consider whether students' mistakes reflect known misconceptions or perhaps previously unknown ones. Teachers may then react to suspected misconceptions with appropriate feedback. All of these tasks are best performed by a human teacher, but a savvy VPS system can also react heuristically to mistakes with feedback.

Teachers and computing education researchers should be alert to, and try to expand their understanding of, how different mistakes in VPS may map to misconceptions. The work we have presented

[11] Our view here is in line with the recommendation of Ben-Ari and Yeshno (2006), who argue on the basis of their empirical study that software should be documented for end users using an explicit conceptual model rather than (or in addition to) task-oriented, minimalistic documentation. This advice seems particularly pertinent when it comes to a VPS system whose purpose is to help users learn about the very conceptual model of a notional machine that the system itself is based on.

in Section 18.2.4 is a step in this direction; many of the mistakes students make with parameter passing in particular (Table 18.2) may well be related to misconceptions about programming concepts.

On 'useless mistakes' and the importance of program animations

Some VPS mistakes are not related to programming, but to failures to find the appropriate GUI operations that match one's intentions. In our study, almost all such mistakes that we could identify had to do with list operations – that is, the new visual elements and simulation steps that the students did not have experience with before.

One way to reduce such mistakes is to improve the usability of the system so that it becomes more natural to use. Another important concern is to ensure that students prepare for VPS assignments. In our data, the mistakes with list operations appeared to afflict especially that large minority of the students who did not watch the program animation of lists in UUhistle. Some pairs only realized too late that they would have benefited from watching the animation, after they had already fought their way through the VPS assignment (see Section 18.2.3).

This result strengthens our belief that it can be useful to intersperse VPS tasks with program animations that serve as worked-out example solutions to VPS assignments. Under most circumstances, such animations should be viewed before doing a VPS exercise on the same topic. Furthermore, students should not be trusted to watch such animations in advance if left to their own devices, without encouragement and a clear incentive to do so. The animations should be presented as a prerequisite for the VPS assignments – this could be enforced by the VPS system. Students might also get a small amount of credit for watching the animations as an additional incentive.

Textual instructions are another way of helping students to find the correct GUI operations and to understand the concepts involved. As we noted in Section 18.2.3, UUhistle v0.2 presented such instructions at the beginning of the first VPS assignment, but most students promptly ignored them. Trying to force students to read lengthy instructions at the beginning of a VPS assignment may not be worth the effort. Instead, advice should be built into the system so that students can access it during the VPS activity in small bits and in a timely manner as the need arises.

Advice from self-explanation studies

It is instructive to view our specific findings concerning VPS in the light of (what are at least intended to be) generic findings from educational psychology.

VPS is a way of presenting example programs to students (Section 14.2). The literature has identified learners' 'self-explanations' of solutions as a substantial factor of the efficiency of examples. There exists a sizable body of research on the use of examples and self-explanation (for reviews, see, e.g., Atkinson et al., 2000; Mayer and Alexander, 2011); this work overlaps significantly with cognitive load research (Section 4.5).

Learners self-explain – explain to themselves – examples in different ways of different quality. Characteristics of good self-explanations of examples have been shown to be: referencing underlying principles; explicit recognition of subgoals and their solutions; anticipation of future solution steps, and critical metacognitive evaluation of one's own understanding of the solutions presented. Unfortunately, learners' spontaneous self-explanations commonly lack these desirable characteristics (see, e.g., Chi et al., 1989; VanLehn, 1996; Renkl, 1997; Pirolli and Recker, 1994).

In a potentially very important study, Renkl et al. (1998) found that spontaneous self-explanations of examples are not as effective as ones produced after a short training period immediately prior to studying the example, during which learners were requested to think aloud and received feedback on their self-explanations from a teacher. This indicates that self-explanation can be taught with reasonable effort. Self-explanation effects and the impact of training on them have also been documented in the context of learning to program (Pirolli and Recker, 1994; Bielaczyc et al., 1995).

Example-based teaching should encourage the recognized good characteristics of self-explanation listed above. Students can be trained in giving self-explanations of examples before they are left to study examples on their own. The advice is consistent with what we have suggested above concerning VPS. The effectiveness of VPS might be significantly boosted if students got personal feedback on think-aloud

VPS work from a teacher near the start of CS1. A single short training session for all or most students might be feasible even in a large-class scenario.

Our advice is also consistent with the related recommendation that the study of examples should be combined with instructional guidance on the content of the examples (see, e.g., Renkl, 2002). Van Merriënboer et al. (2003; van Merriënboer and Kirschner, 2007) recommend that general principles – 'the theory' – are presented before a selection of assignments (such as worked-out examples) on a topic (while also keeping the theory accessible to the student during the assignments). In the context of VPS, training in the principles behind the visualization (the programming concepts) can be provided by teachers or textbooks. Training might also take the form of short, narrated program animations (videos). Researchers warn, however, that merely providing instructional explanations has limited impact on example-based learning, and that eliciting self-explanations may be more important (Wittwer and Renkl, 2010).

18.4.3 We have already incorporated some of our advice into UUhistle

The most recent version of UUhistle (presented in Chapter 13) is different in many ways than the early prototype the students in this study used. The above pedagogical considerations have provided part of the inspiration for some of the new features, in particular the following.

- The way UUhistle gives textual feedback and explains what happens during program execution have been entirely revamped. The Info box in the lower left-hand corner was added to provide feedback at each step of a VPS exercise and to engage students in 'conversations'. The links in the Info box allow the user to ask questions about what is happening and what they should do.

- Since the introduction of the Info box, long textual introductions to VPS assignments have become redundant. Students can access textual explanations as the situation calls for it. Explanations of the visualization and instructions on VPS are now an integrated part of UUhistle rather than separate documents associated with particular assignments.

- Many of the links in the Info box appear in a context-dependent fashion based on simple heuristics. They seek to address potential problems in students' VPS work. Examples of links include:

  - An "I have trouble making progress" link and an associated dialog (see Figure 13.13 on p. 207). These appear if the user makes two or more consecutive mistakes while trying to get past a specific point in the execution sequence, which may be an indication of a naïve trial-and-error strategy (or a difficulty of some other sort). UUhistle tries to guide the learner to reflect on the meaning of the program and question their understanding of it;
  - The "What was wrong with that?" link which appears when UUhistle suspects it may know the reason behind the user's mistake, such as a misconception. The dialogs that the link leads to give feedback tailored to the mistake (see Section 15.3);
  - The "What did I just do?" link that invites the user to reflect on and learn about the meanings of the GUI operations they perform, be they correct or incorrect;
  - A link to a conceptual explanation of the stages of a function call, which in turn links to an operational explanation of how to carry out each of the steps in UUhistle.

- Various small usability tweaks have been made here and there. The "What is this?" links (see Section 13.1.3) allow users to explore the meaning of UUhistle's visual elements, such as the list operators that some of the students in this study confused with each other.

The ultimate VPS system could serve as an intelligent, automatic tutor that is sensitive to students' needs. The features implemented so far in UUhistle are small, tentative steps in this general direction.

Chapter 19

UUhistle Helps Students Learn about What They Actively Simulate

We believe that doing just a single VPS exercise in UUhistle is more trouble than it is worth for students, and long-term use of the system is needed. This belief is supported by the literature on educational visualization. Similarly, studying the use of UUhistle longitudinally could provide researchers with important insights into VPS.

A long-term study was unfeasible for practical reasons. What we were in a position to do, however, was to check for quantifiable short-term improvement in program-reading ability. This chapter reports that empirical work, which has been conducted by the author of the thesis with the help of Lauri Malmi, Kimmo Kiiski, and Teemu Koskinen. We investigated the following question.

> *Does a short VPS session help produce short-term improvement in learners' ability to predict the behavior of given programs?*

More specifically, we looked at two subquestions, using the VPS tool we had available in Spring 2010 (see Section 16.4.2).

1. *Does a short session of studying examples using UUhistle (v0.2) bring about greater short-term improvement in students' ability to predict program output than studying examples without a visualization does?*
2. *Are there differences in the effectiveness of this treatment for different content?*

Our first subquestion suggests an experimental setup, and indeed the way we formulated it is influenced by the quantitative experimental tradition in educational psychology that seeks to measure and compare learning in different groups. Section 19.1 below describes our research methods and experimental setup. Section 19.2 presents the results of our experiment. In Section 19.3, we use qualitative data and a deeper analysis of our test instrument to interpret the quantitative results and answer the second subquestion. Section 19.4 summarizes our findings. In Section 19.5, we consider the implication of our study for the further development of VPS and VPS-based pedagogy. Finally, an addendum in Section 19.6 comments on the effects of VPS on time on task.

19.1 We set up an experimental study to measure short-term learning

We conducted a controlled experiment in which we used a pretest and a post-test to compare the short-term performance improvement of UUhistle users and a control group after a short intervention.

The author of the thesis had the main responsibility for designing the research setup and doing the analysis. Lauri Malmi participated by reviewing the work critically and giving feedback. Two research assistants, Kimmo Kiiski and Teemu Koskinen, organized and supervised the data collection sessions, following instructions from the author of the thesis.

None of the researchers were directly involved in the teaching of the CS1–Imp–Pyth course that the student participants were taking. The two research assistants supervising the data collection sessions were uninvolved in UUhistle's development, but had familiarized themselves with the system.

19.1.1 We recruited a large number of students from a CS1 course

As they participated in the experiment, the students were roughly seven weeks into the spring 2010 offering of CS1–Imp–Pyth (see Section 16.4). They had volunteered to take part after reading an invitation on the course web site.

The students received a fairly small number of assignment points for taking part. The session was advertised (accurately, we believe) as a relatively easy way to gain the points – you would get them by actively participating and would not have to pass any test. This is likely to have biased our sample towards weaker students eager for 'cheap points'; we deemed this to be acceptable.

Students did not know that the research concerned VPS. The invitation explained in general terms that they would participate in a session in which they would practice the topics of the sixth exercise round, and in which data would be collected for the purpose of research on learning programming.

Using an online registration form, each student selected the time they wanted to come to a computer-equipped classroom to participate. A total of 18 sessions were held, with 198 students attending in total. Each session was either a VPS session or a control session, as arbitrarily determined in advance by the session supervisors. The students had no control over which group they belonged to, so the allocation of students to groups was quasi-random. (It is possible that students' schedules introduced some sort of bias, but we are not aware of one and do not expect that any significant bias was created.)

19.1.2 A VPS group used UUhistle, others formed a control group

The sessions took place during the sixth and seventh weeks of CS1–Imp–Pyth. At this time, the main topic the students were dealing with in the course assignments was Python lists. Lists and the associated reference semantics were still new to most of the students: they had had a lecture on these topics (which some had and some had not attended), and some of the students had already done one or more program-writing exercises featuring lists when they participated in the experiment. None of the students had seen any visualizations of lists in UUhistle before.

We first outline the overall structure of both kinds of sessions (VPS and control), then detail the differences between them.

Session structure

The research assistant supervising the session asked the students to form self-selected pairs in which they would work during the session. (The pairing was for research purposes: one of the goals of the sessions was to collect the qualitative data analyzed in the previous chapter, and we wished to get the students to discuss VPS with each other.) Where an odd number of students participated in a session, they formed a group of three students. Late arrivals at a session worked alone. The non-pairs were excluded from the data.

Each session had three phases. The first phase was a pretest consisting of four program-reading assignments. The second phase was the treatment: VPS or control, depending on the session. The third phase was a post-test in which the students were asked to revisit the answers they had given in the pretest.

We wished the sessions to resemble the sort of low-guidance, independent-study situation that large-class courses like CS1–Imp–Pyth routinely place students in. We did not try to provide the students with hands-on guidance from a tutor, however pedagogically justifiable that might have been. Instead, we allowed the participants great freedom in deciding how to approach the example programs they were presented with, much as they would have when encountering a program example or visualization in course materials or a textbook. Specifically:

- the students were free to choose how much time they wished to spend on each phase, and in total. We had allocated a generous amount of time (1 hour 45 minutes) for each session so that (nearly) everyone would have as much time to work on the problems as they wished to spend.

- The students were free to use (or not to use) any learning resources they wished, including lecture notes, course handouts, and other online resources.

- The supervising research assistant did not usually help the students with the problems. The students were asked to try to work on the problems independently but could ask for help with the second-phase assignments if they got really stuck. (Only a small minority of the pairs asked for help.)
- The supervising research assistant did, however, keep an eye on the pairs' progress. In the rare cases where it seemed as if time might run out because the students were stuck, the assistant suggested that they move on to the next phase.

We captured the students' dialogues and computer screens on video. The students were informed of this and all agreed to be recorded. The recording was done primarily for the purpose of the qualitative study presented in the previous chapter, but turned out to be also useful for interpreting our quantitative results; see below.

VPS group: assignments in UUhistle

The VPS group were given three program visualizations within UUhistle v0.2:

- a program animation featuring lists. The animation was prefaced by a textual line-by-line explanation of how the code works;
- a VPS assignment featuring lists and reference semantics. The assignment was prefaced by instructions on how to deal with lists and references in a VPS assignment;
- another VPS assignment featuring lists, references, and function calls.

The code for the three programs appears in Appendix C. At the risk of demotivating the students, we had made the programs 'unplanlike' (they did not match any useful overall goal) so as to require the students to trace the code at a lower level of abstraction. In this early study, we wished to investigate the use of VPS in its 'pure form', and did not include any info dialogs or popup questions in the VPS assignments.

The students were asked to do the VPS assignments. The program animation was suggested to them as a useful resource which they could make use of.

As VPS has a learning curve, the research assistant asked the students whether they had done VPS before during the course. The few pairs in which neither had done so were given the control group task instead and were excluded from our data.

Control group: determine program output without visualization tools

Textbooks and lectures in CS1–Imp–Pyth and elsewhere present example programs for students to look at, mentally trace, and learn from. The student is expected to make the necessary connections between program code, execution-time behavior, and output. We used one form of such tracing-without-visualization activity as the control condition in our experiment.[1] The control group were given the same three Python programs as the VPS group, and similar tasks to do, but no visualizations:

- a program featuring lists. This program was accompanied by a textual line-by-line explanation of how the code works;
- a program-reading assignment in which the students were asked to look at a given program and figure out what it does and why it produces the given output. The program featured lists and reference semantics;
- another similar program-reading assignment featuring lists, references, and function calls.

To avoid bias between the groups, pairs in which neither student had done VPS were also excluded from the control group data. Our results do have the overall bias that all the pairs featured at least one student who had done at least one VPS assignment (as a large majority of CS1–Imp–Pyth students had).

[1] Our intention at this stage was to do a preliminary investigation exploring whether VPS was a feasible approach at all, not to test specific hypotheses related to different modes of visualization use (cf. Section 11.2).

19.1.3 We investigated the improvement between a pretest and a post-test

The pretest consisted of four program-reading assignments featuring lists, reference semantics, and function calls in different combinations. The students were asked to determine the output of the programs. These programs, too, were decontextualized and 'unplanlike'.

The post-test was identical to the pretest in that it featured the same four programs and the students had to revisit their earlier answers. This setup was chosen to ensure that the pretest and the post-test were on exactly the same topics.

The program code of the assignments appears in Appendix C.

Strict assessment: 'perfect' answers required

A program-reading assessment can be assessed in different ways, some of which involve difficult judgment calls of answer quality and challenging concerns of interrater reliability. We wished to be as impartial as possible and avoid such problems. We therefore assessed the pretest and post-test answers strictly, sorting them into 'perfect' ones and 'non-perfect' ones. We considered a 'perfect' answer to be one in which the students had correctly given all the numbers and strings output by the program, in the correct order; mistakes in formatting (e.g., bracketing, whitespace) we ignored.

Measuring improvement between tests

To compare the two groups, we checked whether each pair had produced a 'non-perfect' answer in the pretest but a 'perfect' one in the post-test. This allowed us to compare the distribution of improved and non-improved answers in both populations.

Our data was therefore ordinal and called for non-parametric testing. We used two-tailed Mann–Whitney U tests to compare the groups' answers to each question separately. For each of the four tests, the 'maxed-out' pairs who had already answered the corresponding question correctly in the pretest were ignored, as we were only interested in those students who did not already know the answers to begin with.

We used SPSS Statistics, version 19.0.0, for the statistical tests presented in this chapter.

19.2 We got different results for different test items

Two pairs did not complete the post-test and were not included in the following analysis.[2] This left us with 86 pairs (172 of the 198 students who came to the sessions), 39 in the VPS group and 47 in the control group. As noted, for each question, only those student pairs were considered who did not have a 'perfect' answer to the pretest.

We now present our results concerning the number of student pairs who improved their answers from 'non-perfect' to 'perfect'.

Question 1 dealt with list operations and reference semantics. We found no statistically significant difference between the groups (Mann–Whitney U test, two-tailed, $p = 0.477 > 0.05$). Question 2 dealt with list operations and function calls (including nested calls). In this question, we found a significant difference: the VPS group outperformed the control group ($p = 0.023 < 0.05$). Question 3 featured only a simple function call with list references as parameters; no statistically significant difference was found ($p = 0.478 > 0.05$). Question 4 was about list operations and reference semantics, and again did not produce a significant difference ($p = 0.866 > 0.05$).

Table 19.1 summarizes these findings and details the U values and group sizes for each question. From Table 19.1 it can be seen that for Questions 1, 2, and 4, most students in both groups failed to produce a 'perfect' answer in either the pretest or the post-test. Question 3 most students in both groups already got right in the pretest (and were consequently excluded from the comparison between the groups). The one significant difference we found between the VPS group and the control group concerns Question 2, in which about 28% of the VPS group improved their answer to a 'perfect' one, while the control group showed no such improvement.

[2] One of the two pairs decided they could not "be bothered" to do the post-test. The other pair decided that their time would be better spent going back to thinking about the VPS tasks as "at least from UUhistle we learned something".

Table 19.1: Improvement from 'non-perfect' to 'perfect' answers between pretest and post-test. The pairs who had a 'perfect' pretest answer to a question are excluded from that analysis.

Question	VPS group (39 pairs)			Control group (47 pairs)				
	pairs	improved to 'perfect'	did not improve	pairs	improved to 'perfect'	did not improve	Mann-Whitney U	p
Q1: list operations, reference semantics	37	3	34	46	2	44	883.0	0.477
Q2: list operations, (nested) function calls	29	8	21	33	2	31	581.5	**0.023**
Q3: a simple function call	15	4	11	17	4	13	131.5	0.840
Q4: list operations, function calls, reference semantics	39	2	37	46	2	44	904.0	0.866

19.3 Observations of students' behavior help us interpret our results

What do our results mean as a whole?

Taken by itself, the low p value for Question 2 suggests that students learned better about the topics relevant to that question. Should we conclude that VPS in UUhistle v0.2 appears to improve learning in the short term when it comes to certain topics but not otherwise? Or is the low p value from Question 2 down to chance, an aberration within a big picture that suggests that the UUhistle sessions were not helpful to the students?

We turned to our qualitative data – the video recordings and their analysis from the previous chapter – to help us interpret our results.

To foreshadow what is coming, we believe that the most interesting aspect of our results concerns the contrast between learning about list operations and function calls on the one hand – which the VPS session seemed to help with – and, on the other, learning about references – which our VPS treatment did not help with. Before we get to that, however, we will comment on two other aspects of our results: the low number of 'perfect' answers and the flaws in the design of Question 3.

Few 'perfect' answers overall

The fact that few pairs from either group produced 'perfect' answers to most of the questions is not very surprising. The fact that students struggle with tracing code even after passing CS1 is well established in the literature (see Chapter 3), and these students were only a few weeks into the course. Our decision to look for 'perfect' answers means that our numbers were always going to fail to capture such learning that took place in both groups but did not lead to the ability to trace the programs entirely correctly in the post-test. (That is, we must not conclude that everyone who failed to produce a 'perfect' answer also failed to learn anything.)

That said, we expected the number of 'perfect' answers from both groups to be at least somewhat higher. We list below some other factors that may have reduced the effectiveness of learning within both kinds of sessions.

- The short duration of the interventions may have been insufficient to produce much improvement.
- The sessions were not designed with optimal pedagogy as the foremost criterion. Instead, they were set up, for research purposes, to allow students freedom in their learning efforts, with little or no help from human tutors.
- Asking the students to modify their earlier answers in the post-test was also a pedagogically motivated choice, and was probably a bad idea. In the case of many pairs it was clear that they

did not apply themselves to the post-test, or failed to actively question their earlier thinking on the same problem.

- Many students seemed eager just to finish the session and get the assignment points, rather than take part wholeheartedly. The fact that we actively advertised the sessions as a sure way to get points without having to pass a test probably exacerbated the problem.
- The same advertising probably introduced a bias in our student population towards weaker and less motivated students.
- Neither the VPS assignments nor the corresponding control group assignments had been designed specifically to highlight key issues (because of our decision to investigate VPS in its 'pure form'). Students had to figure out for themselves which were the key moments in each program's execution.
- The artificial, decontextualized nature of the toy programs used will have demotivated some students. (There is some explicit commentary on this in our recordings.)
- The way the assignments were introduced to the students probably failed to emphasize sufficiently the usefulness of reading all the instructions and looking at the given program example. (As noted in the previous chapter, many students failed to study the example and few read instructions, even in part.)
- The programs we used may simply have been too difficult for these students at this time.

Q3 – the flawed third question

In hindsight, we designed Question 3 very poorly. Even a shaky understanding of the constructs used in the program suggests that the program prints out the contents of a list. Given that only two lists are initialized in the code, the question is highly amenable to guessing and accidental 'perfect' answers based on flawed reasoning. Such phenomena may have confounded our test results, and are probably a part of the reason why we got so many 'perfect' answers to Question 3 from pairs who were not able to answer any of the other questions.

Question 3 was also simply easier than the other questions. Some of the things that one does not need to know in order to produce a correct answer to the question are:

- evaluation order within function call expressions: how the value of a composite expression evaluated first (such as another function call or a list initializer) may then serve as a parameter value to a function call;
- variable scope: variables with the same name (but different values) in different scopes, and
- list operations: index-based reading and writing.

Answering Question 2 correctly, however, did require an understanding of all these topics...

Q2 – list operations and function calls: a positive result

The VPS group outperformed the control group in Question 2. A closer look at the questions, combined with the results from the qualitative study that we presented in the previous chapter, gives us an insight into what made Question 2 different for the students.

As noted above, Question 2 involved function calls, nested expression evaluation, differently scoped parameters with the same name, and list operations. Our analysis in Section 18.2.4 suggests that these topics are the ones that students had the most difficulty with during the VPS sessions:

- List operations were a new topic to the students and applying them caused difficulties.
- Parameter passing was a problematic topic. The students had trouble understanding which parameter variables to create and where their values came from. Some of these struggles were:

- attempts to make variables get their values from their namesakes in another frame (rather than from the calling expression);
- being confused by parameter passing when values did not come from a variable in the calling context but from another kind of expression nested within the function call (such as list initialization);
- failures to realize that (or understand why) there can be two variables of the same name in different scopes;
- attempts to resolve invoke a routine before evaluating its parameters when the parameters were complex.

The students' struggles reflect the requirements imposed on them by the particular VPS assignment. Dealing with parameter values in multiple frames, returning values, forming and evaluating nested expressions in the correct order, and setting up list operations are all aspects of program execution that UUhistle's VPS assignments require students to do themselves. To make progress and pass the assignments, the students had to find the operations matching these steps and carry them out in the correct order.

As an additional piece of evidence, we can consider the incorrect answers to Question 2 that students produced. The majority of these were either missing one or both of the last two lines of output, or claimed that the program crashed because of list indexing issues. Even though we are unable to conclusively show the precise reasons behind these incorrect answers, it seems clear that misunderstandings and careless thinking about nested function calls and lists are behind at least some of them.

When we take our quantitative and qualitative results together, they appear to indicate a three-way match between 1) the programming concepts that students have to consider when picking simulation steps, 2) the difficulties that students experience during VPS, and 3) the things students learn the most about during VPS. While we cannot prove a causal relationship, it seems reasonable to conjecture that the students of the VPS group learned better about the topics of Question 2 because those were the things that they had to struggle to address during the VPS task.

Q1, Q4 – reference semantics: no worries, no learning

In contrast with the topics of Question 2, our results suggest that the students did not learn very much about references. Some of the students surely did learn that lists are associated with references, and that such references can be assigned to variables and otherwise used in familiar ways. However, the vast majority of the students failed to learn the main point: multiple references can point at the same object (list) and modifying the object through any one of the references has an effect on all use of the same object. Questions 1 and 4 were effectively impossible to give a 'perfect' answer to without grasping this point. Students' incorrect answers to the questions typically suggested they held the misconception that assignment creates copies of lists. (Other kinds of mistakes were also present, especially in the more complex Question 4, which many pairs got completely entangled with.)

Again, our qualitative data on what the students did during VPS helps us interpret this result.[3]

Reference semantics are conspicuous by their absence from the list of trouble spots of students' VPS work in Section 18.2.4. Indeed, virtually none of the students we observed appeared to have any trouble with creating a copy of a reference to store in another variable, or with using a reference for familiar purposes such as printing.

Why didn't they? Let us take an example. Here is a part of the code of the first VPS assignment, which was intended to teach about reference semantics.

```
second = [3, 1, 3, value]
third = second
second[3] = 100
print second
```

[3]We learned various methodological lessons while conducting this research. One was that we should have put more effort into the design of our test instruments and been more explicit from the start about what *exactly* each question was intended to measure. Another was the concretization of how a mix of qualitative and quantitative analysis can yield interesting insights.

```
print third
second = third
print second
```

And here is a passage that illustrates a fairly typical way of working on this code in VPS:

> *Pair 33 have just reached the line* `third = second`*.*
> **Kate$_{33}$:** Then 'third'... ummm... we make a new one again. And then I do this. *Creates the variable and assigns the reference to it.*
> **John$_{33}$:** Yup.
> **Kate$_{33}$:** And then number three from 'second'.
> **John$_{33}$:** We take the third value... or the third term.
> **Kate$_{33}$:** *Carries out the list assignment.* Yeah. Then we print again...
> **John$_{33}$:** *Small laugh.*
> **Kate$_{33}$:** *In a sarcastic tone:* Exciting! *Carries out the first* `print` *statement.* Okay.
> **John$_{33}$:** All right... and then again, the 'third'...
> **Kate$_{33}$:** So... *Carries out the second* `print` *statement.*
> **John$_{33}$:** [That was] the 'third' value.
> **Kate$_{33}$:** And again we change them.
> **John$_{33}$:** 'second' is changed to 'third'.
> **Kate$_{33}$:** Yeah... *Moves the cursor around the operators area.* Oh yeah, there isn't an equals sign here. Or are we supposed to just drag this over here? *Assigns from* `second` *into* third. No. It can't be this one either, right? *Tries using the assign-to-index operator.* No, did I just do it the wrong way around now?
> **John$_{33}$:** 'third'...
> **Kate$_{33}$:** *Assigns from* `third` *into* `second`*.* That's how it goes, yeah.
> **John$_{33}$:** Yeah.
> **Kate$_{33}$:** And then another 'print'. Yay. *Executes the last* `print` *statement. The end of the program is reached and the assignment is complete.*
> **John$_{33}$:** Yes! Ooh, yeah!
> **Kate$_{33}$:** Yesss! Next example. *Opens the second VPS assignment.*

At no point during the VPS session did Pair 33 pay any attention to the fact that two references point at a single list in the heap. At no point did they pay any attention to how this is demonstrated by the program output (or indeed pay attention to the output at all). Nevertheless, they selected each execution step correctly and eventually completed the exercise without great trouble.

At no point did the VPS exercise require students to understand reference semantics in order to find the correct simulation step. The students had no trouble with references not because they understood the concept well but because there was no particular reason for them to pay attention to it. Most of the students we observed simply ignored the topic entirely or took it for granted.

In the light of the above, and of existing research supporting active learning, it seems reasonable to conclude that a significant part of the reason why VPS did not help students learn much about reference semantics lies in how the VPS assignments did not *require* them to use their knowledge of the topic.

Other factors may also have contributed to students' lack of attention to references. The visualization of references in UUhistle v0.2 (see Figure 16.1 on p. 262) is not very attention-grabbing. The toylike nature of the example programs may have made program output (which highlights the impact of reference semantics) less interesting for the students. In a single case (described in Section 18.2.3 above), we saw a pair consciously decide against believing the system's visualization of references.

Despite the problems, some students did pay attention to references. Pair 6, whose work on the same code is described on page 307, is one of these rare cases.

19.4 VPS helps with – and only with – what learners focus on

Let us try to summarize our findings. Our experimental study enables us to answer the question: "Does VPS in UUhistle v0.2 work?" with a cautious "Yes, but..." The system appeared to help students learn

to predict program behaviors better than merely studying the examples without the help of a visualization did, but the help did not apply equally to all kinds of programs. Our study highlights the importance of the second of our research questions – "Are there differences in the effectiveness of this treatment for different content?" – and suggests that the way a VPS assignment engages (or fails to engage) students with their intended object of learning is of crucial importance.

In our study, VPS helped the students learn about what they themselves had to think about to pick the correct simulation steps. It did not help them to learn about reference semantics, a topic passively present in the VPS exercises but not central to the students' task. That is, the students appear to have learned about precisely what the requirements of the VPS activity made focal, and not much else.

This interpretation of the results is both intuitively appealing and a good fit with the existing literature. The human tendency to focus – for better or worse – one's visual attention and mental efforts exclusively on task-related issues is well documented in the psychological and educational literature.[4] Not only did the VPS task not appear to help students notice the way multiple references may point at the same object, and the consequences of this fact, but it is also quite possible that a focus on finding the correct VPS step actually detracted from the attention students paid to the relationships between references and the objects they point to.

Our results show only a modest increase in collective ability to deal with Question 2 effected by the VPS treatment. When reading the results, however, we must realize that as many students make limited progress during an entire semester of CS1, we cannot expect immense gains from a single very short session. The results are also likely to have been affected by some of the specific features of our experimental setup, as discussed at the beginning of Section 19.3. In any case, the most significant outcome from this study lies, we believe, not in measuring how much of an improvement an experimental session with a VPS prototype produced, but in the insights the results give for the development of better variants of VPS and educational software visualization in general.

Our results must be interpreted cautiously. Our study suggests that VPS is better than the alternative of studying given examples without visualization, but we cannot claim an improvement to other alternative treatments that VPS might have been compared against, including other forms of program visualization and how-to-write-it guidance. Having only investigated the short-term effects of a few VPS assignments given in the middle of CS1, we also cannot comment on how VPS impacts development of students view of programs overall during the very first weeks their programming studies. Furthermore, both our experimental and control conditions were low on tutor guidance, to match the independent work setting of the CS1 course we studied. VPS with more tutor guidance might work differently but would of course have to be judged against a different control condition.

Ultimately, the value of VPS is determined by its effects on learning programming as a whole and in the long term. Our short experiment is only a small step towards better understanding the educational impact of interactive program visualization.

19.5 The study suggests that VPS works, but should be improved

We are reasonably satisfied with the impact of VPS in our study. Even though we used an early prototype, and although the use of UUhistle in the course had overall been more 'strapped-on' than integrated with the other teaching, we observed a positive short-term transfer effect from a short VPS session.

Even so, our study clearly indicates that we must look at ways of improving VPS and its implementation in software systems. In particular, our study has highlighted the need for extreme care in the design of VPS assignments. In order for VPS to work, the system and assignment designers must consider each interaction that the students are expected to make and the degree to which those interactions – and not the other content of the visualization – match the intended learning goals. Otherwise, students may end up performing trivial or inapposite operations that do not teach them what was intended.

[4] For example, Simons and Chabris (1999) document the extent of our 'inattentional blindness' to anything other than what we are currently absorbed in, even if it is spatially close to what we are focusing on. Suthers and Hundhausen (2003) showed that having people construct a particular sort of visualization for a collaborative task led them to focus their knowledge-building discourse on exactly those aspects that the visualization made relevant. Marton and Booth (1997) reviewed studies that indicate how presenting students with questions about a text in advance led them to focus exclusively on what seemed to be immediately relevant to answering the questions.

The previous chapters have already presented pedagogical suggestions for improving VPS and its use in teaching in general. We extend this advice below as we consider, by way of example, how to teach better about reference semantics in a VPS context.

Reference semantics: two general observations

Before we present concrete suggestions for improving VPS, we have two general points to make regarding how reference semantics relate to VPS exercises.

First, in VPS, the student takes on the role of a notional machine. However, as the notional machine (or a concrete runtime) makes copies of reference values and as it dereferences them, it is not concerned with which other references may point at the same data, and does not access those other references. This aspect of reference semantics is therefore not a natural part of the decision-making process of VPS.

Our second, related point concerns the degree of learner engagement with a visualization. As described in Section 13.6, UUhistle's VPS exercises seek to engage students on the applying level of the 2DET engagement taxonomy: the students apply a given visualization to show what happens within the notional machine. However, when it comes to the impact of reference semantics on program behavior (as in the program on page 330 above), the VPS exercises do no more than a program animation does. The user does not directly simulate the carry-on effects of modifying an object on multiple sections of code, they only observe those effects happening by paying attention to the visualization and the program output (if indeed they do, that is). With respect to this topic, a VPS exercise is in fact not applying at all but merely controlled viewing with a complex user interface that draws attention to itself.

Drawing attention to references

For students to learn better about reference semantics in a VPS context, or program visualization more generally, two kinds of improvements may be useful. First, and perhaps more importantly, assignments should require students to engage with the topic on a level higher than controlled viewing. Second, VPS systems and assignments should be designed so that they seek to draw students' attention to the impact of references on program behavior.

What follows is a list of specific techniques that might be used for these purposes. Some of them concern references in particular, while others are more generic.

- Explanatory materials could be embedded into VPS exercises to draw students' attention to important issues and avoid key execution steps from getting lost in the crowd. For instance, a popup dialog or further info link might appear to explain how an object being modified through another reference has affected an output.

- Similarly, popup questions and hybrid assignments in which students only simulate a crucial section of code could highlight important points and require students to engage with them. For instance, students might be asked to specify how many objects have been created and how many references point to each at a given moment in time, or to explain (by selecting from multiple choices) why the two lines of output they just produced were identical.

- Specifying the output of a `print` statement could be made a part of the student's task (instead of it working as a black box).

- Although the machine is not concerned with keeping track of *which* references point to an object, it may be concerned with *how many* there are. A different kind of VPS assignment might involve reference-counting-based garbage collection.

- Various tricks could be employed to draw visual attention to references. For instance, arrows can be used in various ways to highlight even references that are not directly involved in the current operation, although care must be taken to avoid visual clutter. Having program output glide into the I/O console might help to get students to notice it. Different forms of highlight could be used to signal about changes in memory during the previous execution step.

- The use of programs with a domain familiar to students and/or with a clearly defined purpose (as opposed to arbitrary manipulations on lists of integers) could help draw students' attention to program output and the way reference semantics affect it.

UUhistle supports the use of popup dialogs and hybrid assignments (see Chapter 13). The newer versions also feature a number of additions (which v0.2 did not have) that seek to help the student notice the importance of references. These include improved visualization of references (Figure 13.1, p. 194), better tooltips, and context-sensitive links in the Info box (e.g., Figure 13.4, p. 198). It remains to be judged whether any of these gentle, unintrusive measures have a tangible impact. The way forward may lie in requiring students to engage more with visualizations of references.

19.6 Addendum: VPS helped by increasing time on task

Time on task is a significant factor in any pedagogical technique, with software visualization being no exception. One of the problems with example-based learning is that students may not apply the necessary attention to (and spend the necessary time with) the examples, especially in courses such as CS1–Imp–Pyth where all attendance and participation is voluntary. One of the hypothesized effects of VPS (see Chapter 14) is that it increases students' time on task as they study program examples, compared to merely giving students example programs to study from or to other, more passive uses of program visualization. This is because VPS requires the students to take a detailed look at programs that is assessed at each step by the VPS system.

Beyond our main research questions, our experimental setup allowed us to explore the effect of VPS on time on task. Our sessions allowed students to choose freely how much time to spend on the intervention between the pretest and the post-test. From our session recordings, we checked the length of this interval for each pair (rounding the start and end times to the nearest minute, and computing the difference).

The VPS group used a mean time of 28.3 minutes (median 25.0, standard deviation 13.9). The control group used a mean time of 15.5 minutes (median 14.0, standard deviation 5.67). The difference between the groups is statistically very significant (independent-samples t-test, $N = 88$, $t(51.6) = 5.56$, $p \approx 0.000$, equal variances not assumed).[5]

Our result supports the notion that VPS leads students to spend more time and cognitive effort on the visualization and/or the example program. We conjecture that this increased time spent explains in part the better learning in the VPS group.

Time on task is, of course, not a goal in itself, but must be considered in terms of bang for buck compared to alternative pedagogies. Further study is needed to better evaluate the relationship between improvements in learning outcome through VPS and time on task.

[5]The data was not normally distributed, and had a high kurtosis of 9.38. However, given the high sample size, a t-test was appropriate.

Chapter 20
The Students Liked It

Well, more or less.

Ideally, learning is both fun and effective. However, student opinions are a notoriously treacherous measure of the effectiveness of a pedagogy. Many designers of educational SVs, for instance, have ruefully observed that students generally respond to graphical representations of software by saying that they "liked" the visualization or that it "was good", yet it is much less clear whether those visualizations have resulted in significant learning gains.

People like it when there is clarity and certainty, but clarity and certainty are sometimes the result of failing to leave one's comfort zone. Meaningful learning is often troublesome and discomforting as it requires the learner to challenge their existing ways of thinking (see, e.g., Chapter 9 on threshold concepts). Sometimes, it is the discomfited learner who has made more progress than the one who "likes it". Negative correlations have been found between student enjoyment of various teaching methods and learning achievement (Clark, 1982). Learners' assessments of what they themselves know are known to be unreliable (Kruger and Dunning, 1999).

As long as one is aware of these caveats, student feedback can be a great source of information for the educationalist. Students' opinions can highlight the strengths and weaknesses of a pedagogical approach and suggest improvements. Affective factors are important and affect students' motivation. Even if students liking a pedagogy does not guarantee effectiveness, or their disliking it mean the approach is a failure, it is important to explore and reflect on the reasons behind students' responses. Making everyone have fun all the time as they learn is not possible, but unnecessary problems should be addressed.

In this chapter, we review feedback on UUhistle assignments from CS1 students in order to explore the research question:

How do students react to the use of UUhistle in CS1 (and why)?

Beyond studying students' emotional response to UUhistle, our review of course feedback serves a triangulatory role as a supplement to the studies in the previous chapters.

The feedback review was conducted by the author of the thesis with the help of Lauri Malmi as a critical discussant. The data comes from the course 'CS1–Imp–Pyth' described in Section 16.4.

The chapter consists of two sections: in Section 20.1, we present our review, and in Section 20.2 we consider the implications of our findings.

20.1 CS1 students answered a feedback survey

Near the end of the spring term, CS1–Imp–Pyth students are invited to answer an extensive feedback survey. Participation is voluntary, but students get a small number of assignment points for taking part. Recently, the survey has featured several multiple-choice and open-ended questions about the UUhistle assignments.

Three points concerning the survey need to be made. First, due to the timing of the survey, students who have dropped out tend not to answer, which introduces a bias to the results. Second, the questions in the survey were about the program-reading assignments in UUhistle in general, including both the program animations and the VPS assignments. The results below are a response to the use of UUhistle

in the course more generally, not only to VPS. Third, not all students answered all the questions in the survey; the questions on UUhistle were probably only answered by those respondents who had done at least some of the UUhistle assignments (most, but not all did; see Section 16.4). The opinions of those students who chose not to do the UUhistle assignments at all (or perhaps failed to notice their existence) are therefore probably not well represented in the results below.

20.1.1 The response to UUhistle was mixed, but more positive than negative

Figure 20.1 presents the distributions of students' answers to six UUhistle-related multiple-choice questions. These results are from the same spring 2010 offering as the studies in the previous chapters, detailed in Section 16.4. (The answers to another survey in spring 2011 followed a similar pattern.) The other CS1 offerings in which UUhistle's VPS assignments have been used did not have a directly comparable survey – more on those courses below.

Most students rated UUhistle's usability positively (Figure 20.1a).

Students' assessments of statements of the form "The UUhistle assignments helped me [with X]" (b through e in Figure 20.1) were mixed. Many agreed with each statement, while many others disagreed with it. Most students agreed at least somewhat that UUhistle helped them learn about the execution order of programs (b). Students were somewhat less convinced that UUhistle had helped them learn about computer behavior (c) or to understand program code outside the system (d). Less than half of the students agreed with the claim that UUhistle helped them write programs of their own (e).

The survey asked students to choose a statement that corresponds to their idea of how suitable UUhistle is for the course, in principle and in its present incarnation. The distribution of answers (f) again indicates that there is a spread of opinions, with no single dominant view. The majority of students thought UUhistle is useful, at least in principle. However, many felt that significant improvements to the system or assignments would nevertheless be appropriate.

20.1.2 Open-ended feedback highlights strengths and weaknesses

We now review the answers from students' answers to open-ended feedback questions in course-end surveys. The quotes below come from the CS1–Imp–Pyth offerings in the springs of 2010 and 2011, and from similar CS1s offered as self-study courses during the summer months of the same years. In the surveys, students were asked to "describe, in a concrete way, something that you learned in UUhistle", to comment on the pros and cons of UUhistle, and to suggest improvements. Our data also incorporates some open-ended feedback voluntarily sent by students as they were submitting an assignment.

Let us set the scene with a selection of quotes that illustrates the broad spread of opinions and emotions within the student population. We will then comment on salient themes we identified within the open-ended responses.

> It was easy to use, the assignments were nice and since it was useful to do them (I learned something new and started to understand coding a bit better), my motivation to spend time on the assignments grew. I'd have liked to have assignments about every topic on the course. I also did all the voluntary bonus assignments, because it was fun!

> [I learned:] Everything! Sure, I learned a bit at the lectures, "a-ha, that's how you read the code", but in UUhistle I SAW that "a-ha, this is how the code really works"! When you know perfectly what order the computer reads your code in, it's easy to write, too! Even though many of my friends dissed UUhistle, for me it was the best thing in the course. Don't drop it!

> THAAANK YOOOU! I had never before been interested in programming, so I know next to nothing about it. Now I have got the feeling that I too have been taken into consideration in this course, and they've managed to make a proper beginner-friendly example so that I too will understand.

> With the help of UUhistle, you got to understand the logic of the coding language, which helped you build your own programs. A concrete example is better than a hundred lectures about the topic.

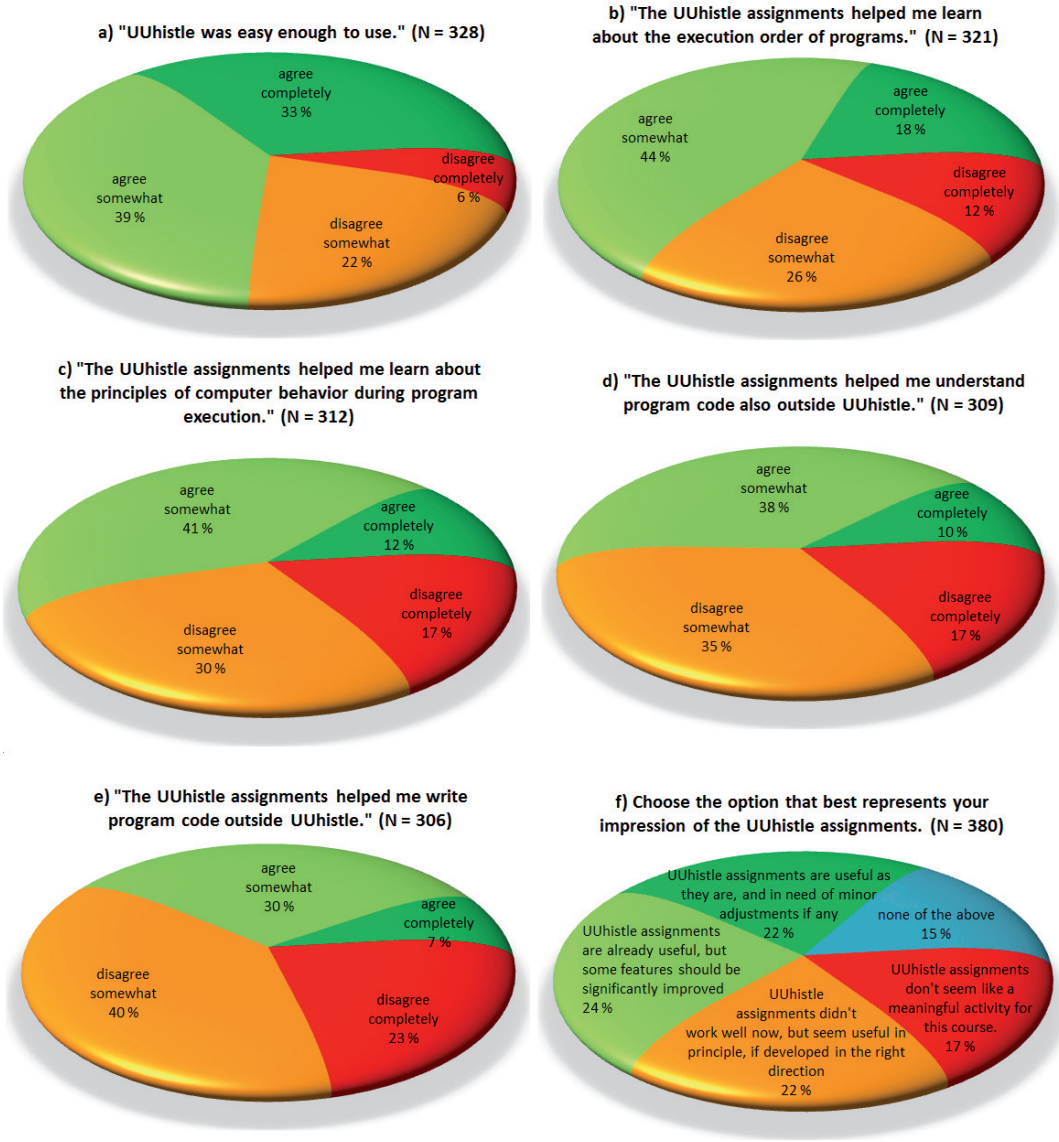


Figure 20.1: Students' answers to multiple-choice questions in a feedback survey. Each of these questions also had a "no comment" option; those answers – and blank answers – are not included. The N values indicate the number of people who answered one of the other options shown.

> Good both as an idea and in functionality. Definitely supports learning if you put thought into doing the assignments. Occasionally there was too much repetition in the programs and the programs weren't always about meaningful/sensible topics.

> Even though this wasn't my first programming course, only through the UUhistle assignments did I really understand how and why even the simple actions that you program work.

> UUhistle is, for sure, useful to people with no experience of coding at all. Since this wasn't my first coding course, the UUhistle assignments were a bit annoying. Maybe make UUhistle voluntary somehow, or something?

> A bit unclear here and there, but nevertheless a nice way to get to know a new subject.

> A good idea and would be a wonderful learning support, but often remained somehow distant and didn't manage to really clarify things all that much.

> [I learned:] Nothing, really. An annoying clickfest and I usually got what a piece of code does without its help.

> It's hard to comprehend something like this, where you do nothing sensible! No more of this!

> I can't think of any suggestions for improvement, but I thought that programming itself was easier than using UUhistle, even though many of my friends said that UUhistle was really useful. Sometimes I got stuck in a dead end in UUhistle and couldn't figure what to do and the instructions didn't help. The only suggestion for improvement that I can think of is to improve the instructions that you get, especially while the program is running; the instructions at the beginnings of assignments were actually quite good.

> You learn to code by writing code, not by dragging boxes across the screen.

> [I learned:] How much something can get on your nerves.

> Absolutely awful crap, I hope I never have to have anything more to do with the system in question.

> A good psychological exercise, doing the assignments tests your ability to tolerate something that's fucking maddening.

Learning about general topics: execution order and a new way of thinking

Dozens of students singled out the execution order of programs as something they had learned about in UUhistle. Here are a couple of typical examples:

> Especially the order in which different instructions are executed (for example in function calls) was greatly clarified thanks to UUhistle.

> The order that things get done in within the computer became more tangible.

Another popular theme was that UUhistle had helped bring about a new perspective on programs. Many students attributed to UUhistle a newfound understanding of the computer's 'way of thinking' and/or of program execution. For instance:

> [I learned:] The "way of thinking" of the computer or its logic.

> [I learned:] Perhaps that everything really does advance only step by step and not predictively somehow.

> I learned to understand "the logic of Python", I mean, what the program does and in what order and so on.

> UUhistle taught me how the computer "thinks". My understanding at least was greatly increased by seeing in a concrete way how a program works (how the machine interprets the code). For instance, the calling of functions and object-oriented programming were much clarified through UUhistle.

> It clarified the material that we had to learn about. It made you think "inside out".
>
> In UUhistle, I learned about how programs work. I learned how information flows within "programs". Concepts like "variable" and "function" got partially new meanings through UUhistle.
>
> [I learned:] Computer thinking doesn't work like mine.

The prominence of these two kinds of answers in the survey results may have been influenced by the presence of the multiple-choice questions of Figure 20.1 in the spring course questionnaires. However, those questions were not present in the summer course surveys and the same kinds of answers were nevertheless common there too.

Learning about specific topics: functions and objects

Students brought up various specific topics as ones that the UUhistle assignments had helped them learn about. Two answers were markedly more common than the others: functions and objects. A few examples for each topic are given below.

> Only with the help of UUhistle did I get how to use functions.
>
> [I learned:] The execution order and hierarchy of functions, because you don't necessarily see that when you execute the code.
>
> Functions spring to mind first, starting values [parameters] and return values, and maybe loops.
>
> The True function and the False function [sic] and the use of functions in general, because in the assignments you had to jump backwards again and again.
>
> [I learned:] What things require a new worktop [a new frame, presumably]. Where the stored information is located and how it's found.
>
> The most help I got from UUhistle was when I was trying to figure out references in object-oriented programming.
>
> [I learned:] how to use objects. The lecture notes clarified this fairly well, too, but in practice it was clarified best through UUhistle.
>
> In object-oriented programming it was pretty good and gave concreteness to a topic that was otherwise a bit abstract.
>
> Programming objects would have been much harder without UUhistle. You don't learn about them only from using UUhistle, but UUhistle has a different perspective that was really useful for me at least.
>
> How objects work became clearer in UUhistle. It was far easier to do object-oriented programming with the help of UUhistle than it would have been if I hadn't done the UUhistle assignments.

Other topics mentioned by smaller numbers of students included variables, conditionals, loops/`while`, lists, recursion, nested function calls, and references.

Some students reported that UUhistle had been particularly good for learning the basics during the early rounds, but that the later UUhistle assignments had been too long, difficult or otherwise unrewarding. Some other students felt the opposite and stated that the early assignments had been too trivial while the later ones were useful.

Putting the lessons learned into practice

Many students reported that they had been able to take something from the UUhistle assignments and apply it to program-writing, program-reading, or debugging tasks outside of the system.

> Very easy to use. Demonstrates in many cases very precisely how the code works which makes it easier to understand and write code.

> In many cases the UUhistle assignments taught me how to do all the assignments of a round without having to read the lecture notes.

> The demos in UUhistle were useful because you got a good initial understanding of how new things work in Python, and you didn't have to start programming all unprepared.

> [I learned:] How the computer processes information and how I myself have to think and what I have to do to make the computer behave correctly and the way I want.

> [I learned:] In some things, the execution order of a program, which led me to figure out how to write one of the [program-writing assignments].

> I liked UUhistle because it sort of showed how the program thinks and that helped me understand why I sometimes got the [program-writing] assignments wrong.

> [I learned:] To notice and fix special cases that happen when programs are run.

> I always did the UUhistle assignments last so that I wouldn't make any mistakes in them. This is sure to have made the regular [program-writing assignments] harder to do, but served as an excellent recap of the new things that I had learned so they stuck in my mind better.

A number of responses mentioned that simply seeing example code had been useful (but the visualization perhaps less so). For instance:

> [I learned:] Models (wholes) for the [program-writing] assignments

> The left-hand side [where the program code is] sometimes had useful code structures for the other assignments.

> You could copy code and edit it into your own program.

> I can't remember anything specific, but with some topics I always had the UUhistle assignment open as I was writing code for the other assignments of the same round. Sometimes the code in UUhistle gave good pointers on what you should write yourself.

Irrelevance to course or programming in general

Many students also stated that UUhistle had not been useful, complaining that they had not learned anything or had only learned to use UUhistle. For instance:

> [I learned:] Nothing, a completely terrible system!

> I learned what kind of program one should never create.

> [I learned:] to try everything.

> [I learned:] to use the UUhistle program.

> [I learned:] To press buttons in the correct order.

Some students explicitly pointed out that they perceived little or no connection between what they did in UUhistle and the goals of the course.

> The UUhistle assignments were, with a few exceptions, only training in the use of UUhistle (i.e., mechanically clicking through the assignment) that had no connection to actual programming.

> I learned the internal logic of UUhistle, but little about programming itself.

> [I learned:] To press the forward button as quickly as possible. I didn't see any connection between UUhistle and the [program-writing] assignments or writing code.

> At first, I didn't even understand what UUhistle had to do with the course. I don't think that it clarifies or helps you to program the course assignments.

> I don't know if it's appropriate, in a basics of programming course, to cover in so much detail what happens "behind the scenes", so to speak.

Some students requested better integration of the UUhistle assignments into the rest of the course.

> The idea is good, but UUhistle should be better related to coding itself. For instance, so that it would be possible to see what the code looks like in its entirety.

> It would be good to shed some more light on "UUhistle's world" at the beginning of the course.

> The lectures/labs should clarify – a little bit at least – what the point of the system is.

> UUhistle was somehow really unclear as a whole. I might even suggest that the UUhistle assignments confused some of the things I learned. I never discovered any logic in them. Also, towards the end of the course I forgot to do many weeks of UUhistle assignments, [since] they weren't emphasized in the course at any point. I didn't even get what they were, why we had to drag boxes from one place to another.

Unfair demands regarding execution order

A few students commented negatively on the idea that UUhistle enforces a particular way of simulating program behavior. One student was very upset about this:

> I learned that you can solve them in precisely one way and in precisely the order that the assignment creator has planned. This was the most frustrating thing about the whole course. [...] It was wretched how you could only do the operations in a specific order in one single way. A good example of this was mathematical calculations (their order sometimes didn't make any sense). This gave a disgusting impression of what programming is.

The student's comment suggests that he or she does not recognize a well-defined order for the evaluation of arithmetical expressions is appropriate or necessary. The student appears to attribute the order to UUhistle rather than to the Python language and runtime.

Trial and error

A number of students noted that it was possible to score points from the UUhistle assignments mechanically without putting thought into the matter, and emphasized the responsibility of the students' own mental involvement.

> You can score points without thinking for a moment about what you are doing.

> Only useful if you wanted it to be. Often I'd just click through it without caring one bit about what it said.

> UUhistle made you really think about what the code causes to happen, but sometimes you'd just do the assignments in the hope of scoring points if they became too difficult.

> The assignments could be solved by just trying things, even without understanding the principles behind the assignment. On the other hand, the UUhistle assignments were clearer and more meaningful to do than the [program-writing assignments] and I always found more motivation to do the UUhistle assignments, and tried nevertheless to understand what I did in UUhistle as best I could.

> UUhistle helped to make sense of things. It still wasn't especially useful, but that was due to myself since it was pretty easy to score full points for the assignments without thinking at all, and I just wanted to get the assignments done quickly.

Boredom and excessive detail

A fairly common complaint about UUhistle was that the assignments were boring, too long, and too detailed, and/or involved too many clicks of the mouse.

> The UUhistle assignments were boring and you'd just try to do them as quickly as possible by trying everything.
>
> The UUhistle assignments were boring. You'd need to introduce some more creative element to them.
>
> Sometimes pretty boring to go through, I liked writing programs myself more.
>
> Some assignments demanded that we grind through the same loop over and over again, which was frustrating. I think less repetition of routine things would have been enough to make things clear. Now the repetition led to frustration which ate away at the potential usefulness of the other parts of the assignments.
>
> If you already knew the subject, the assignments were painfully slow to do.
>
> The bad thing was that all the moving of numbers to the right boxes in the right order was done in too much detail, the long way.

In this matter, as in practically all the matters present in the feedback, contradictory opinions were also given, with a student stating, for instance, that the UUhistle assignments "were nice and pretty quick to do".

Program animation vs. VPS

In many of the survey responses quoted above, it was unclear to what extent the complaints about excessive length or tedium were about the program animations and to what extent they were about the VPS assignments. Some of the complaints were clearly about the animations. Some students explicitly contrasted the two usages of UUhistle, usually in favor of the VPS assignments.

> Requires a great deal of motivation from the student. A motivated person will surely learn with the help of the program, but the rest of us just keep hitting the forward button to score points.
>
> The annoying thing was long programs where you only had to watch the next step for 10 minutes...
>
> Long assignments where you just watch the computer execute the program encourage you to just set it to full speed and click without thinking.
>
> Remove all the "watch how this goes" assignments, they are boring and don't teach anything. Have more assignments where you make the program execute yourself, they are much better for learning.
>
> The automatized ones I couldn't be bothered to think about at all.
>
> I just fast-forwarded through the assignments where UUhistle did everything for me to get the free points as quickly as I could. This didn't support my learning, of course, but perhaps those assignments shouldn't be worth any points since that's precisely the kind of behavior that it leads to, I expect. The simulation assignments were excellent and very instructive.

The opposite opinion was also expressed, and a few students suggested that alternative uses of UUhistle would be better than the VPS assignments.

> UUhistle is unwieldy and doesn't necessarily correspond at all to how many people perceive programming-related topics. Only the automated examples were worth the time spent watching. The simulation assignments on the other hand wasted time since it was too slow and unintuitive to simulate simple processes in UUhistle by moving boxes and the like. Just learning to use UUhistle took nearly as much time as doing the actual simulation assignments.

> A better way could be to simulate code partially automatically and occasionally ask the user what the code returns or something like that.
>
> [I learned:] Nothing. I literally don't understand why such a system has been created. It would be much better if they were just examined during the lectures instead of doing all this drag-and-drop foolishness.

In fall 2010, the author of the thesis incorporated UUhistle's program animations into another CS1 course that used the Java programming language. Animations were used extensively in lectures and there were some small program-reading assignments in which students would watch animations in UUhistle and answer occasional popup questions. The course – let us call it CS1–OO–Java – used a 'quick and dirty' modified version of UUhistle that animated teacher-configured examples written in a subset of Java. VPS was not used because the implementation did not support it for Java programs. Two of the students who took CS1–Imp–Pyth in summer 2011 commented on the differences between their experiences of UUhistle in CS1–OO–Java, which they had taken earlier, and the VPS assignments they now encountered for the first time in CS1–Imp–Pyth. Both had a favorable opinion of VPS:

> It was totally awesome that in this course you had to do something yourself in UUhistle, and not just look at what happens or try to figure out what a function prints out (this was new compared to the Java course).
>
> In the UUhistle assignments, it was good how in some of them you had to know how to do the next step yourself. In the basic course with Java, the UUhistle assignments were just click-to-advance assignments, so you didn't learn nearly as much from them.

Insufficient guidance

UUhistle's user interface received both praise and criticism; students disagreed amongst themselves as to whether the program was "easy to use", "clear", "in need of some fine-tuning", "usually clear", "confusing", "unclear", or "unintuitive". A recurring cause for complaint was lack of guidance when the user made a misstep.[1]

> If you got stuck, it was really hard to figure out how to go forward.
>
> UUhistle could explain in more detail why a mistake happened and not just say that it was a mistake.
>
> A downside was that when you were stuck, nothing helped except trying a thousand things.
>
> In some problematic situations it could give some hints about what has gone wrong, although you can of course just find the answer by trying everything.

Some students were frustrated by the difficulty of figuring out what to do in UUhistle, even though they felt they knew what was supposed to happen next in the program.

> Sometimes it was hard to make progress, even though you knew what would happen next, when the program expected me to click some other button instead, or to do things in a slightly different order.
>
> Sometimes I got stuck for a bit when I didn't know where to click even though I more or less knew how to proceed based on the code
>
> I understood the code of the program, but it still took an eternity to figure out how to drag those pieces, what a waste.

[1] Comments in this vein were more common in spring 2010 than in the most recent CS1–Imp–Pyth offerings. The issue appears to have been somewhat alleviated by the introduction of the Info box and the gradually increasing amounts of guiding material in later prototypes.

A handful of students suggested that doing the VPS assignments was in fact more difficult than doing the program-writing assignments, or was predicated on concepts that you only learn from writing programs.

> I never really got the hang of how UUhistle works. Even though I easily knew how to do all the programming assignments of a round, I'd still not get how UUhistle works.
>
> UUhistle is quite okay in principle. But you often don't really understand the things you do there unless you can write the corresponding things into code. And if you already can, does UUhistle serve a purpose any longer?

Assignment design

A few students complained about the inauthenticity of the program examples given in UUhistle.

> The programs in UUhistle sometimes didn't seem to have any connecting thought to them. I mean, they would execute, but the reason and the purpose were left in the air.
>
> The UUhistle assignments were good, with a few exceptions. Sometimes it remained unclear what a program is supposed to do: I mean, not how it works but the logic of what it's intended for.
>
> The UUhistle assignments mostly just made me even more confused. I didn't feel that I got anything out of them. I can't really say how they could be improved; maybe making them resemble real code more would help?

A few also regarded the assignments as too easy and suggested that more challenging assignments be used instead or in addition to the UUhistle assignments used.

> The assignments in UUhistle could be a bit harder. I mean, UUhistle could show more complicated structures than those that were included now.
>
> Most of the UUhistle assignments were laughably easy 'for dummies' exercises. On the other hand, I suppose this may have been the idea, and may in fact have helped a bit with getting started with the assignments, especially as one doesn't have earlier experience of programming.

20.2 Student feedback has been consistent with the findings of the previous chapters

We summarize below some of the main implications that our review of course feedback results suggests to us.

- Overall, UUhistle is usable enough to be useful, but there is room for improvement.
- Some students appear to have been greatly helped by UUhistle. Others appear not to have been helped at all. Many others fall somewhere in between.
- VPS brings many students to focus on execution order. Further, it helps students develop a new way of thinking about programs in terms of their execution by a computer.
- VPS in UUhistle is perhaps particularly useful for learning about function calls and objects.
- Some students failed to understand a purpose to UUhistle at all, or at least to connect what they learned in UUhistle to other course content. On the other hand, others successfully and eagerly transferred what they learned in UUhistle to their programming activities.
- Some students largely ignored the dynamic visualization aspect and used UUhistle primarily as a repository of code examples.

- Students get frustrated when they fail to make progress during VPS and sometimes resort to trial and error. UUhistle's in-built support in such situations is not yet satisfactory.
- A significant number of students found VPS to be tedious. A perhaps larger number of students felt the same about program animations. Some of the stronger and more experienced students were put off by the 'trivial' nature of the VPS assignments (which were identical for all students and not open-ended).
- The above points appear to have contributed to many highly emotional responses to UUhistle, with some students enthusiastic about the system and others very negative.

These points are consistent with the results of the studies from the preceding chapters. In particular, students' opinions of VPS and their descriptions of what they learned are compatible with the categorization, in Chapter 17, of the distinct ways in which students understand what it means to learn though VPS. The problem of some students not being able to relate the visualization to programming concepts and the rest of the course content is prominent in the survey results, as in the previous chapters.

The survey results give additional evidence of the strong emotional responses students have to VPS in UUhistle. As noted in the introduction to this chapter, such responses need interpretation. The fact that some students do not 'like' UUhistle or that they find it troublesome or annoying to deal with is not *necessarily* a problem in itself. To the extent that it is caused by the inherent troublesomeness of being forced to deal with difficult subject material, not liking VPS is acceptable from an educational perspective. However, the affective aspects of the approach must certainly not be ignored. We must consider what reasons there are behind students' comments and seek to address any outstanding issues. In our case, it appears that several factors contribute to the unenthusiastic responses of some students. Perhaps the most important are limited understandings of the purpose of VPS, the repetition of steps within assignments, and the lack of sufficient guidance when 'stuck'.

The results presented in this chapter further support the pedagogical strategies and directions for VPS system design that were outlined in Chapters 17, 18, and 19. The integration of VPS with other materials and teaching is crucial to encourage a rich learning experience and to give a positive impression of VPS. Assignments that focus on a key section of the code could help reduce tedious repetition and highlight key content. Careful selection of a rich array of examples can help demonstrate why expression evaluation order is so important and why it plays such a key role in VPS. Allowing and helping students to visualize and debug their own programs in a system such as UUhistle may guide them to perceive the meaning and potential of the visualization and add to the value of VPS. Advice and hands-on guidance from teachers and teaching assistants is needed to allow as many students as possible to benefit from VPS. Future VPS systems could be increasingly intelligent, with ever more guidance built into them.

Finally, we note that in the course we investigated, all the students were encouraged equally to do the UUhistle assignments, and the assignments contributed to all the students' grades. Educational research has shown that the effectiveness of many instructional strategies depends dramatically on the prior knowledge of each learner (see Section 4.5). VPS may not be for everybody. In a more flexible setting, especially in courses with fewer students, teachers might be able to suggest VPS assignments specifically to those students who appear to need them most. Students who have no trouble with function calls, expression evaluation, and the notional machine, for instance, might be better off spending their time on other kinds of assignments.

Part VI

Conclusions

Introduction to Part VI

In Parts I, II, and III, I drew on the literature to identify challenges in introductory programming education, explored these challenges in the light of several theories of learning, and reviewed approaches to teaching introductory programming courses. From this review, the issue of program dynamics and the notional machine stood out as one of the major challenges in the failure of introductory programming courses around the world to work as well as intended. In Part IV, I suggested as a solution the pedagogical approach of visual program simulation, which was subsequently evaluated in Part V. It is now time to reflect on what has been accomplished, and what may be accomplished in the future.

Part VI consists of two chapters. Chapter 21 summarizes the main conclusions and contributions of this thesis. Chapter 22 rounds up this book with a peek into the future.

Chapter 21

Visual Program Simulation is a Feasible Pedagogical Technique

In this thesis, I have investigated visual program simulation, a pedagogical technique for introductory programming education. In VPS, a student takes on the role of the computer as executor of a given program. The student uses a visualization of an abstract computer, a 'notional machine', as an aid to illustrate what the computer does as it processes the program. The goal of the VPS activity is to help the student learn about programming in general and about specific programming concepts.

The thesis contributes to VPS in several ways. First, I have formulated the idea of visual program simulation and provided a theoretical framework for it. Second, I have presented a software system that facilitates the use of VPS in practice. Third, I have described preliminary empirical evaluations of VPS and the software. Fourth, I have made recommendations for pedagogy and VPS tool design that arise from the empirical work. The following sections, 21.1 to 21.4, deal with these four aspects of our work. In Section 21.5, I briefly consider the broader implications of my thesis.

21.1 VPS is a theoretically sound pedagogical approach to teaching programming

VPS has a solid foundation in multiple learning theories. In Chapter 14, I related VPS to the various learning theories and instructional approaches reviewed earlier in Part II.

Computer programs exist as static program code and as dynamic entities at runtime. This duality is such an obvious part of practitioners' tacit knowledge that it is often barely acknowledged as important; nevertheless, the dynamic aspect is very challenging for some novice programmers to grasp. In cognitive constructivist terms, students need to form a viable, robust mental model of the notional machine. In Chapter 14, I argued that VPS can help learners discern the crucial dynamic aspect to programs and programming, and to construct a better mental model of the mechanisms underlying program code. In this way, VPS can help novice programmers across a key early threshold. Practice using VPS serves to ingrain the mental model and make it a natural, efficient part of the learner's thinking.

Computing education research suggests that learner interaction is an important factor in the educational effectiveness of a visualization. VPS builds explicitly on this idea: the learner is not allowed to be a passive consumer but must be an active user of the visualization. This increases the likelihood of the learner cognitively engaging with the visualization and the underlying concepts. VPS as presented essentially fuses together two ideas: the kind of program visualization that is commonly used in program animation tools, and interaction in the form of direct manipulation of a visualization. In the context of programming education, direct manipulation had previously mostly been used within higher-level algorithm visualization and in visual programming.

Misconceptions and limited understandings of programming constructs are a common cause of problems and frustration to novices. Feedback on an incorrect simulation step during VPS can engender fruitful cognitive conflict and help learners develop rich, viable understandings of programming concepts.

Finally, VPS is a program-reading activity in which the student traces the execution of a given program rather than creating a program of their own. Reading code is increasingly being recognized in the literature

as an important component of introductory programming education, to be learned before or in parallel to program writing. A program-reading task such as a VPS exercise can be used to provide an interactive worked-out example of program writing; judicious use of examples helps manage learners' cognitive load during the learning process.

21.2 VPS can be made practical with a tool such as UUhistle

A software system empowers visual program simulation. Automation makes possible continuous availability to large groups of students, instant feedback, and automatic grading. Moreover, it brings the convenience of an already-implemented visualization which learners apply to simulate programs, and makes it easier to gain practice from multiple VPS exercises.

This thesis contributes the conceptual design of a prototype software visualization system. The system, UUhistle, supports VPS (among other modes of program visualization) on a particular notional machine for the Python programming language. UUhistle visualizes the state of the notional machine as abstract graphics, and allows the user – the learner – to directly manipulate the graphical elements so as to demonstrate how the computer executes a given program. The system has been implemented and has been made available to the computing education community at `http://www.uuhistle.org/`.

UUhistle is not a finished article. We have sought to critically evaluate our prototype to learn more about what works and what does not, in UUhistle as well as in VPS more generally.

21.3 VPS helps students but there is room for improvement

Part V of this thesis contributes a preliminary empirical evaluation of VPS in the context of an introductory programming course. The findings from this mixed-methods research project suggest that VPS is a promising pedagogical approach that has helped many university students to learn programming. At the same time, the evaluation highlights certain weaknesses in the approach that must be taken into consideration when adopting VPS.

The phenomenographic study of Chapter 17 discovered several qualitatively different ways in which programming students perceive what it means to learn through visual program simulation. A rich understanding of VPS opens up rich possibilities for learning programming, in keeping with the goals outlined for VPS in terms of learning theory. In such a rich understanding, VPS is understood as being related to how computers execute programs, and that in turn is understood to relate to how one reads and writes computer programs as a programmer. Other, more limited ways of understanding, in which these crucial aspects of learning through VPS are not discerned, significantly limit the effectiveness of VPS. In order for VPS to fulfill its potential, care must be taken in teaching to help learners develop a rich understanding of what VPS is and how it can help them.

Another exploratory study in Chapter 18 contributes a concrete view into VPS sessions, illustrated through quotes relating to significant episodes observed in actual practice. The study concretely demonstrates instances of learning within our proof-of-concept VPS system. In addition, the study concretizes how students use various kinds of information to find the correct execution steps during VPS. Although, from a pedagogical point of view, students ideally reason about the conceptual content of the visualization, trial-and-error tactics and guesswork based on superficial visual features also exist in actual practice. Typically, students use a mix of tactics; the important question is perhaps not whether superficial tactics ever feature in a student's VPS work, but whether the work is characterized overall by a deep approach to VPS that looks for meaning in the visualization and the selection of simulation steps. It is such a deep approach that instruction should seek to foster. A preliminary analysis of student mistakes in the same chapter serves as an example of how VPS can serve as an aid for teachers and researchers who wish to understand what students think about programming concepts.

Chapter 19 reports an experimental study of the effects of a short VPS session. The findings suggest that the short-term effectiveness of VPS is dependent, perhaps greatly, on precisely what students have to reason about as they choose simulation steps. Our study highlighted in particular how a VPS treatment was helpful in teaching about expression evaluation and function calls – which UUhistle's VPS assignments require the student to carry out in detail – but not about reference semantics – which were present in the

assignments but not focal to user actions in the same way. This result shows that extreme care must be taken in the design of VPS systems and specific assignments so that required user interactions are aligned with intended learning goals.

The review of course feedback in Chapter 20 is in agreement with the findings of the other studies. It further highlights the strong emotional responses – both positive and negative – of some students to the UUhistle system. The results suggest that the guidance students get after they make a mistake is decisive and must be improved in UUhistle. This study affirms the conclusion that a concentrated teaching effort is needed to make the purpose of VPS clear to students. Assignments must be carefully designed so that they are not overly repetitive and consequently boring. According to student opinions, UUhistle works best when it comes to learning function calls and object-oriented programming, two topics that are recognized as difficult in introductory programming education. A minority of students complained about usability issues but the majority found UUhistle to be sufficiently easy to use.

21.4 VPS needs to be designed and used thoughtfully

Visual program simulation is a tool for meticulous instructional design. It is a potentially useful weapon in the CS1 teacher's arsenal, but it is a weapon that needs to be sharpened and handled with care. The empirical research presented in this thesis suggests that VPS must be used and developed in a considered way for it to be as useful as possible to as many learners as possible.

A good introduction to VPS is important. VPS should be clearly introduced as an activity that is about computer behavior and that serves the student as they participate in the course and learn programming. If students get the point of VPS early and are motivated to learn from it, they avoid wasting time and are in a position to make the most of VPS. A good introduction is unlikely to be enough, however. The relationships between VPS and other aspects of the course – classroom instruction, program-writing assignments, textual materials – are crucial. VPS is not going to be maximally effective as an isolated component of a course that students are expected to connect to the rest of the material themselves. For best results, VPS should be integrated into the learning process in a concrete and explicit way.

Our results suggest that it is important for teachers (and VPS systems) to observe how their students make decisions regarding simulation steps and how (or whether) they reflect on what they do. Through observation, the students can be led to use VPS effectively. Teachers and materials should encourage students to think conceptually about the content of the visualization and to reflect on what they are doing and what they have done. Strategies such as trial and error should be discouraged as pedagogically ineffective; nevertheless, occasional use of such tactics is not a problem if instruction has fostered a sufficiently deep overall approach to VPS, and students are likely to reflect on what they have accomplished afterwards. The early adoption of a deep approach to VPS is crucial so that students' subsequent VPS activities (even when unguided by a human tutor) will be motivated by a search for meaning.

Both students and teachers can learn from students' VPS mistakes – teachers may find VPS useful as an analytical tool that helps them elicit students' understandings and misconceptions and to address them. This aspect of VPS may also turn out to be useful for research purposes.

Successful assignment design is key to unlocking the potential of VPS. Assignment design involves many considerations, including program complexity and authenticity, required user interactions, degree of repetition, level of abstraction, and of course the learning goals of the assignment. Our empirical findings highlight the particular importance of the relationship between learning goals and the user–system interactions required to find the correct simulation steps: VPS users are likely to learn about the specific things that they consider when picking simulation steps (and perhaps not much else). A judicious combination of different program visualization techniques and modes of interaction – VPS, program animation, popup questions, info dialogs, etc. – may allow the best opportunities for aligning learning goals with user interactions, and consequently produce the best results.

Some of these recommendations for teachers can be worked into future VPS systems, which would be a significant boon in large-class courses where the availability of human guidance is limited. In Part V, we have made recommendations for the design of VPS systems along these lines. In the spirit of action research, we have already started to improve UUhistle in line with the results from our empirical work.

In Appendix D, I have concretized our main pedagogical advice regarding VPS in bullet point format.

21.5 The contribution of the thesis may extend beyond VPS

The work I have presented in this thesis may have implications beyond its focal point of visual program simulation. For instance, the findings on the ways of experiencing learning through VPS (Chapter 17) may well be applicable to other forms of educational program visualization in introductory programming education, and to the educational use of visualizations in general. Just providing a visualization system and saying a few words about it is unlikely to produce optimal results. Learners need to be helped to perceive meaning in the tools they are given, and to link that meaning to the intended learning goals. The importance of specific user interactions on the educational impact of VPS (Chapter 19) may also have significance not only for VPS but for the designers of other interactive learning tools. Ultimately, the portability of my research must be assessed by its readers.

The thesis contributes a fairly broad review of the literature on teaching and learning introductory programming, and the challenges concerning program dynamics in particular. Among other things, this review instructs the programming teacher that whether one adopts VPS or not, program dynamics and the notional machine should be explicitly dealt with when setting learning goals for introductory programming courses and when designing teaching and learning activities to match those goals.

A research-methodological contribution may be seen as another outcome of this thesis. Computing education and computing education research continue to have a lot to learn from other fields, such as psychology and education. While theories from those fields are increasingly being used within computing education research, many research projects build – for reasons practical or parochial – on a single theory or perspective. This thesis contributes to the computing education research community an example of theoretically grounded pedagogical design that is informed by multiple complementary theories and research traditions. The mixed-methods empirical evaluation of the resulting pedagogical approach is similarly ecumenical. Irrespective of any particular merits and shortcomings of the present work, it would be pleasing to see more multiparadigmatic studies in the computing education research of the future.

Chapter 22

What is Next for Visual Program Simulation?

In this chapter, I briefly sketch out some possible futures for visual program simulation. I list some ideas on future research on the current incarnation of VPS (Section 22.1), on tool development (Section 22.2), and on how VPS could be used in different ways and in different contexts (Section 22.3).

22.1 There is plenty to research in VPS as presented

The most obvious follow-up to the present work is 'more of the same'. The effects of VPS on learning programming could be qualitatively and quantitatively investigated in more course offerings and by other researchers. Future work may critique and perhaps improve the warrants for our claims about VPS, and expand on those claims. Long-term studies would be especially useful for evaluating the overall impact of VPS on learning programming. The aspects of VPS that we have explored qualitatively could be investigated quantitatively; follow-up studies could determine the frequencies of students' approaches to and notions of VPS, for instance.

A potentially very fruitful source of data is the mistakes students make during VPS. Students' mistakes could be categorized from automatically generated session logs and analyzed both qualitatively and quantitatively to discover what kinds of mistakes are common and to explore the causes behind them. The work we have initiated in mapping students mistakes to misconceptions (in Chapter 18) could be extended. Such work could have implications not only for the use of VPS but for introductory programming education more generally. We already have a sizable database of students' VPS logs; a master's thesis that investigates this data is being written.

A future study could compare the impact of VPS to another form of visualization use such as controlled viewing. Such studies could be structured in terms of the 2DET taxonomy, a side product of this thesis from Section 11.2.3, and might contribute towards establishing a more generic framework of learner engagement in software visualization.

In the course offerings we have investigated, all students were given the same VPS assignments to do, irrespective of prior knowledge or other factors. Documented phenomena such as the expertise reversal effect (Section 4.5) suggest that prior knowledge has a dramatic impact on the effectiveness of learning activities. Further research is needed on the interactions between VPS, learners' programming expertise, and learning outcomes. This line of research is important in order to understand how to use VPS in classes in which students' prior knowledge is highly variable, and also in order to attend, in instructional design, to the growth of expertise during learning.

Cognitive load theory (Section 4.5) suggests still more avenues for future research. Cognitive load measurement is a very demanding but potentially very worthwhile endeavor (Paas et al., 2003; Plass et al., 2010). The effects of VPS on different types of cognitive load are presently unknown. Knowing more about them would further our understanding of the impact of VPS on learning, aid tool development, and allow us to position VPS exercises more accurately in relation to more familiar types of learning tasks such as worked-out examples and problem-solving assignments (cf. Section 14.2).

This thesis has grounded VPS in a number of relevant theories from different research traditions, but

the picture is hardly complete. Future studies could examine VPS from perspectives yet different, such as collaborative and social learning, process-to-object learning (Sfard, 1991), multimedia learning (Mayer, 2005, 2009), and the paradigm and language wars within the programming education community, to name but a few. Studies could also explore the relationships between VPS and comparable simulation activities in related fields such as engineering education.

Another possible research direction concerns the role of teaching staff in the use of VPS, and the ease or difficulty of successfully adopting VPS as a pedagogical approach. The impact of training teaching assistants to help students develop a rich view of VPS could be investigated, for instance.

22.2 There are tools to be developed

VPS relies on software. Continued constructive research on VPS systems is essential for the potential success of VPS as a pedagogical approach. Such research can involve the development of existing systems to support VPS better or the creation of new systems.

One obvious path for improving on the current version of UUhistle is to extend its support for the Python language. Supporting a language fully or near-fully is useful in a program visualization tool especially if it is to be given to students to animate and debug their own programs. Support for other programming languages could also be added.

The ultimate VPS system of the future would adapt to each student's needs. In Part V, I have suggested specific ways in which a VPS system could be 'smarter' and more sensitive to the learner's needs and current context. Work is ongoing to continue beyond the tentative first steps we have taken towards providing context-sensitive, misconception-aware feedback to the learner. Increased dialogue between student and system about programming concepts and the reduction of trial and error are other goals for the future. Moreover, VPS assignments – like other forms of example study – are not equally useful to all students. One key issue is the learner's growing (and prior) expertise. Rapid diagnostic methods (see Plass et al., 2010; Kalyuga, 2009) might be incorporated into visualization systems to assess students' knowledge online and to tailor the use of VPS and other materials in response to individual differences.

The usability of UUhistle is acceptable, but hardly perfect. Formal usability studies could be conducted to evaluate the system and suggest improvements. One known concern is the need for the user to repeat routine steps in more complex assignments; this issue should be addressed with a combination of usability improvements, assignment design, and increased use of abstraction.

Future VPS systems (and program visualization systems in general) could visualize execution on different notional machines and at different levels of abstraction than the current UUhistle does. An intriguing but challenging challenge would be to develop support within the same system for multiple different levels of abstraction between which the user can change, even during a single program run. A system might provide, say, a high-level object-interaction view and a low-level bytecode view to a program run in addition to an intermediate level such as the one now displayed by UUhistle. Such a system could provide a very powerful way of illustrating, juxtaposing, and learning about different perspectives on program dynamics.

Auditory presentation of some sort could be integrated into VPS in order to make better use of learners' dual-channel processing and to optimize the use of precious working memory capacity (Mayer, 2005, 2009).

The integration of VPS into a system that visualizes roles of variables and/or uses metaphorical visualization elements (cf. pp. 164-166 above) might be worth exploring.

UUhistle is currently a stand-alone program; support for IDE integration could allow it to be used more conveniently in some contexts.

The configuration of VPS assignments in UUhistle currently involves the editing of raw XML files. A better configuration interface should be provided for the benefit of teachers and developers. Teachers could also be supported with other conveniences for creating assignments in UUhistle, such as a facility for creating popup questions whose answers are determined automatically by the system (e.g., "What is the highest number of frames simultaneously on the stack during this program run?"; cf. Rößling et al., 2011).

Implementing these ideas into tools calls, of course, for further evaluative research.

22.3 VPS can be taken to new users and new modes of use

Last but certainly not least, the future of VPS will be shaped by its future users, the contexts in which they use it, and the new uses they may find for VPS.

Pedagogical practices, especially ones that require work from the teacher, are difficult to disseminate. VPS can serve as a useful component of a carefully crafted CS1, but it does not offer a silver bullet that effortlessly solves students' or teachers' problems. It would be unrealistic in the extreme to expect VPS to be a likely 'instant hit' in the mainstream of CS1 education. A more realistic expectation is that some programming teachers who are concerned about issues regarding program dynamics and the notional machine – and who have enough time to follow computing education research and actively develop their teaching – could work VPS into their courses. Some of those teachers may develop their own assignments. Some may offer assignments for others to use. Scholars within the software visualization community may implement VPS or something inspired by it into their program visualization systems (as is already happening in the ViLLE system; see Section 11.3). Through new users and their input and experiences, VPS may gradually grow and improve.

A broad selection of good ready-made assignments would be an excellent asset in the dissemination of VPS to the community. I expect to develop and publish a set of assignments for UUhistle in the not-too-distant future.

Many of the VPS assignments in our studies were unplanlike, which limits their usefulness as worked-out examples of program writing (Section 14.2). The combination of VPS with more planlike programs should be explored in the future. VPS could also benefit from what is known concerning the use of examples in learning; training students in explaining VPS assignments to themselves is one potentially rewarding strategy (see Section 18.4). Instructional design models such as 4C/ID (Section 4.5) have the potential to inform the design of introductory programming courses and to clarify the role of VPS in such designs.

In the future, VPS can be used in new combinations with other materials. Hybrid program visualization assignments (featuring animation, VPS, and quizzes) are one such combination. The integration of VPS and other modes of program visualization into hypertextbooks is another (cf. Shaffer et al., 2011). Motivating new contexts for VPS might also be found, for instance by having students simulate buggy programs.

VPS has been designed primarily to target CS1 courses, but could serve a purpose in some slightly more advanced courses as well. At Aalto University, a data structures and algorithms course has already taken the initiative to adopt VPS for a few assignments, with staff reporting largely positive experiences.

Again, all these ideas for the future provide ample opportunities for empirical research.

"Show, don't tell" is the advice they give to aspiring fiction writers. What they mean is that you should not directly describe a character's feelings or thoughts. Instead, you should let the reader get involved and work things out for themselves from how the character talks and behaves. "Show, don't tell" also reflects the thinking behind much of visualization-based education, although there the phrase tends to be taken more literally: use pictures rather than words.

Why bring this up? Perhaps it is better, dear reader, that I do not tell you.

Part VII

Appendices

Appendix A

Misconception Catalogue

> *On two occasions I have been asked, – "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?"* [...] *I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.* (Babbage, 1864, p. 67)

This appendix lists, based on the literature, examples of novice programmers' misconceptions about the content of introductory programming courses. The misconception list is drawn from exploratory research, most of which is qualitative. The appearance of a misconception in the list below is not a statement on its commonness. Section 3.4 above presented a short review of the studies on which this appendix is based.

"Misconceptions" or what?

Most of the novice understandings listed below explicitly go against the technical definitions and the "accessible ontological reality" (see Section 6.7) of the computer. Some others are not so much contradictory with reality as they are partial: the full extent of a concept definition is not grasped. Yet other items in the list correspond to vague "difficulties" with topics that have been mentioned in the misconceptions literature but whose exact character is unclear.

This is a list of not only apples and oranges, but also of tomatoes and the odd dried plum. I have contrived to place side by side results from a theoretically, methodically, and terminologically disparate set of studies that spans several decades. What I have lumped together as "misconceptions", the original researchers have variously called "misconceptions", "partial understandings", "incorrect understandings", "student-constructed rules", "difficulties", "mistakes", "bugs", and so forth. Some of these differences in terminology are superficial, others are motivated by learning theory – some phenomenographers, for instance, adopt a perspective in which poorer ways of understanding concepts can be seen as limited (rather than mistaken) versions of richer understandings (see Chapter 7).

All of the items listed nevertheless have something in common: they describe difficulties with content commonly covered in CS1 courses.

What is not listed?

I have left out issues that appear to be highly language-specific or tool-specific as well as trivial mistakes concerning notation such as mistaking an operator for another. Some misconceptions do appear in the list that are particular to a currently popular CS1 programming language or a group of similar languages.

I have excluded topics that are often not explicitly covered in CS1, such as concurrency and definitions of program correctness.

Even though program design and domain modeling are central to numerous CS1 courses, I have left out higher-level design issues and focused on coding and program execution, which relate more closely to the topic of this book.

On Table A.1

In Table A.1, topics of misconceptions are roughly grouped into one of *General* (the overall nature of programs and program execution), *VarAssign* (variables, assignment and expression evaluation), *Control* (flow of control, selection and interation), *Calls* (subprogram invocations and parameter passing), *Rec* (recursion), *Refs* (references and pointers, reference assignment and object identity), *ObjClass* (the object–class relationship and instantiation), *ObjState* (object state and attributes), *Methods* (issues specific to methods and methods calls), *OtherOOP* (other topics specific to object-oriented programming), and *Misc* (none of the above). The categories (like the misconceptions) are fuzzy and overlap, but I have nevertheless only assigned a single topic to each misconception. This categorization is only meant to give some structure to the list; the labels are not meant to be taken taxonomically.

The descriptions in Table A.1 have been paraphrased from the original sources; some are my abstractions from multiple similar findings. I have included a few items marked "local", which are based only on my own casual observations or those of my teaching colleagues.

Table A.1: Novice misconceptions about introductory programming content

No.	Topic	Description	Source
1	General	The computer knows the intention of the program or of a piece of code, and acts accordingly.	Pea (1986)
2	General	The computer is able to deduce the intention of the programmer.	Pea (1986)
3	General	Values are updated automatically according to a logical context.	Ragonis and Ben-Ari (2005a)
4	General	The system does not allow unreasonable operations.	Ragonis and Ben-Ari (2005a)
5	General	Difficulties understanding the lifetime of values.	du Boulay (1986)
6	General	Difficulties with telling apart the static and dynamic aspects of programs.	du Boulay (1986); Ragonis and Ben-Ari (2005a)
7	General	The machine understands English.	du Boulay (1986)
8	General	Magical parallelism: several lines of a (simple non-concurrent) program can be simultanenously active or known.	Pea (1986)
9	VarAssign	A variable can hold multiple values at a time / 'remembers' old values.	du Boulay (1986); Soloway et al. (1982); Putnam et al. (1986); Sleeman et al. (1986); Doukakis et al. (2007)
10	VarAssign	Variables always receive a particular default value upon creation.	du Boulay (1986); Samurçay (1989)
11	VarAssign	Primitive assignment works in opposite direction.	du Boulay (1986); Ma (2007); Putnam et al. (1986)
12	VarAssign	Primitive assignment works both directions (swaps).	Sleeman et al. (1986)
13	VarAssign	Limited understanding of expressions which lacks the concept of evaluation.	local

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
14	VarAssign	A variable is (merely) a pairing of a name to a changeable value (with a type). It is not stored inside the computer.	Sorva (2008); Doukakis et al. (2007)
15	VarAssign	Primitive assignment stores equations or unresolved expressions.	Bayman and Mayer (1983); du Boulay (1986); Putnam et al. (1986); Doukakis et al. (2007); Sorva (2008)
16	VarAssign	Assignment moves a value from a variable to another.	du Boulay (1986)
17	VarAssign	The natural-language semantics of variable names affects which value gets assigned to which variable.	Putnam et al. (1986); Sleeman et al. (1986); du Boulay (1986); Kaczmarczyk et al. (2010)
18	VarAssign	The order of declaration of variable names affects which value gets assigned to which variable.	Sleeman et al. (1986)
19	VarAssign	There is no size or precision limit to the values we can store in a programming variable.	Doukakis et al. (2007)
20	VarAssign	Incrementing a counter variable is an indivisible operation (no separate evaluation of right-hand side).	Soloway et al. (1982)
21	VarAssign	Primitive types (in Java) have no default value.	Kaczmarczyk et al. (2010)
22	VarAssign	Unassigned variables of primitive type (in Java) have no memory allocated.	Kaczmarczyk et al. (2010)
23	Ctrl	Difficulties in understanding the sequentiality of statements.	du Boulay (1986); Simon (2011)
24	Ctrl	Code after `if` statement is not executed if the `then` clause is.	du Boulay (1986)
25	Ctrl	`if` statement gets executed as soon as its condition becomes true.	Pea (1986)
26	Ctrl	A false condition ends program if no `else` branch.	Putnam et al. (1986); Sleeman et al. (1986)
27	Ctrl	Both `then` and `else` branches are executed.	Sleeman et al. (1986)
28	Ctrl	The `then` branch is always executed.	Sleeman et al. (1986)
29	Ctrl	Using `else` is optional (the next statement is always the `else` branch)	Sleeman et al. (1986)
30	Ctrl	Adjacent code executes within loop.	Putnam et al. (1986); Sleeman et al. (1986)
31	Ctrl	Control goes back to start when condition is false.	Putnam et al. (1986); Sleeman et al. (1986)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
32	Ctrl	Difficulty in understanding automated changes to `for` loop control variables.	du Boulay (1986)
33	Ctrl	`while` loops terminate as soon as condition changes to false.	Pea (1986); du Boulay (1986)
34	Ctrl	`for` loop control variables do not have values inside the loop or their values can be arbitrarily changed.	Putnam et al. (1986); Sleeman et al. (1986)
35	Ctrl	Print statements are always executed, irrespective of branching statements.	Sleeman et al. (1986)
36	Ctrl	All statements of a program get executed at least once.	Sleeman et al. (1986)
37	Ctrl	Subprogram code is executed according to the order in which the subprograms are defined.	Ragonis and Ben-Ari (2005a); Sleeman et al. (1986); Vainio (2006)
38	Calls	A return values does not need to be stored (even if one needs it later).	Hristova et al. (2003)
39	Calls	A method can be invoked only once.	Ragonis and Ben-Ari (2005a)
40	Calls	Numbers or numeric constants are the only appropriate actual parameters corresponding to integer formal parameters.	Fleury (2000)
41	Calls	Difficulties distinguishing between actual and formal parameters. Confusion over where parameter values come from.	Hristova et al. (2003); Ragonis and Ben-Ari (2005a)
42	Calls	Difficulties understanding the invocation of a method from another method.	Ragonis and Ben-Ari (2005a)
43	Calls	Confusion over where return values go.	Ragonis and Ben-Ari (2005a)
44	Calls	Parameter passing forms direct name-based procedure-to-procedure links between variables with the same name (in call and signature).	Madison and Gifford (1997)
45	Calls	Parameter passing forms direct procedure-to-procedure links between variables with different names (in call and signature).	Madison and Gifford (1997)
46	Calls	Parameter passing requires different variable names in call and signature.	Madison and Gifford (1997)
47	Calls	Subprograms can (routinely) use the variables of calling subprograms.	Fleury (1991)
48	Calls	Cannot use globals in subprograms when they have not been passed as parameters.	Fleury (1991)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
49	Calls	When the value of a global variable is changed in a procedure, the new value will not be available to the main program unless it is explicitly passed to it.	Fleury (1991)
50	Calls	A function (always) changes its input variable to become the output.	Paz and Leron (2009)
51	Calls	Expressions (not their values) are passed as parameters.	local
52	Calls	Argument expressions cause changes in existing variables.	George (2000a)
53	Calls	Upon return, the value of a variable changes to correspond to match a previously given parameter.	George (2000a)
54	Rec	Null model of recursion: recursion is impossible.	Kahney (1983)
55	Rec	Active model: only the 'active aspect' of recursion understood, not return values.	Götschi et al. (2003)
56	Rec	Step model: single- or two-step recursion, but no more.	Götschi et al. (2003)
57	Rec	Return value model: each instantiation first produces a value, before the next one is started; then all values are combined to get the result	Götschi et al. (2003)
58	Rec	Passive model: active part not understood, only the combinatorics of the return values.	Sanders et al. (2006)
59	Rec	Recursion is merely a programming construct used as a template in certain kinds of programs; runtime behavior is 'magic'.	Booth (1992); Kahney (1983); Götschi et al. (2003)
60	Rec	Recursion is perceived as an algebra problem.	Götschi et al. (2003)
61	Rec	Looping model of recursion (single-instantiation); recursion is merely a construct for producing repetition, not understood as self-reference.	Kahney (1983); Bhuiyan et al. (1990); Götschi et al. (2003); Booth (1992)
62	Refs	Even primitive values (in Java) are handled through references.	Sorva (2008)
63	Refs	The variables for storing and assigning primitive values are fundamentally different from the variables used for storing objects (in Java).	Sorva (2008)
64	Refs	A variable associated with an object is merely a name which can be used to manipulate an object or 'thing' in a program.	Sorva (2008); Ma (2007)
65	Refs	A (non-primitive Java) variable does not hold a reference but a set of object properties.	Sorva (2008); Ma (2007)
66	Refs	Assigning an object means 'sending' an object (or a copy) to a variable.	Ma (2007)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
67	Refs	Assigning to an object causes it to become equal to the assigned object.	Ma (2007)
68	Refs	Assigning to an object means that (some) instance variables get new values from the assigned object.	Ma (2007)
69	Refs	Confusion between an instance variable such as "name" and a variable referring to the object.	Holland et al. (1997)
70	Refs	Two objects with the same value for a "name" attribute are the same object.	Holland et al. (1997)
71	Refs	A value of an instance variable such as "name" replaces reference as a value of an attribute.	Sajaniemi et al. (2008); Ragonis and Ben-Ari (2005a)
72	Refs	An object is represented (in program state) by only the value of a particular identifying instance variable such as "name".	Sajaniemi et al. (2008)
73	Refs	A variable that refers to an object uniquely specifies the object for all time.	Holland et al. (1997)
74	Refs	Once a variable references an object, it will always reference that object.	Holland et al. (1997)
75	Refs	Two different variables must refer to two different objects.	Holland et al. (1997)
76	Refs	Two objects of the same class with the same state are the same object.	Holland et al. (1997)
77	Refs	Two objects can have the same identifier if there is any difference in the values of their attributes.	Ragonis and Ben-Ari (2005a)
78	Refs	Objects know what refers to them. References point in opposite direction (to referring object attribute, not from there).	Holland et al. (1997); Sajaniemi et al. (2008)
79	ObjClass	Confusion between a class and its instance.	Holland et al. (1997)
80	ObjClass	An object is a subset of a class. / A class is a collection of objects.	Teif and Hazzan (2006); Ma (2007); Détienne (1997); Ragonis and Ben-Ari (2005a); Teif and Hazzan (2006); Vainio (2006)
81	ObjClass	An object is a subtype of a class.	Teif and Hazzan (2006)
82	ObjClass	A set (such as "team" or "the species of birds") cannot be a class.	Teif and Hazzan (2006)
83	ObjClass	Constructors can include only assignment statements to initialize attributes.	Fleury (2000); Ragonis and Ben-Ari (2005a)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
84	ObjClass	Instantiation involves only the execution of the constructor method body, not the allocation of memory.	Ragonis and Ben-Ari (2005a); Kaczmarczyk et al. (2010)
85	ObjClass	Difficulties understanding the empty constructor.	Ragonis and Ben-Ari (2005a)
86	ObjClass	Initializing an attribute with a constant as part of its declaration causes confusion in distinguishing between a class and an object.	Ragonis and Ben-Ari (2005a)
87	ObjClass	Initializing an attribute with a constant within the constructor declaration causes confusion in distinguishing between a class and an object.	Ragonis and Ben-Ari (2005a)
88	ObjClass	Invocation of the constructor method can replace its definition.	Ragonis and Ben-Ari (2005a)
89	ObjClass	Difficulties in understanding where objects of a simple class are created before the creation of the object of a composed class.	Ragonis and Ben-Ari (2005a)
90	ObjClass	Difficulties understanding objects if their attributes are not explicitly initialized.	Ragonis and Ben-Ari (2005a)
91	ObjClass	Objects are created by themselves, without explicit instructions to create them.	Détienne (1997); Ragonis and Ben-Ari (2005a); Kaczmarczyk et al. (2010)
92	ObjClass	Declaring a variable also creates an object.	Ma (2007)
93	ObjClass	You can define a non-constructor (Java) method to create a new object.	Ragonis and Ben-Ari (2005a)
94	ObjClass	A textual representation of an object is a reference to the object.	Holland et al. (1997)
95	ObjClass	There is no need to invoke the constructor method, because its definition is sufficient for object creation.	Ragonis and Ben-Ari (2005a)
96	ObjClass	If objects of a simple class already exist, there is no need to create the object of a composed class that builds on them.	Ragonis and Ben-Ari (2005a)
97	ObjClass	If the object attributes are initialized in the class declaration, there is no need to create objects.	Ragonis and Ben-Ari (2005a)
98	ObjClass	The creation of an object of a composed class automatically creates objects of the simple class that appear as attributes of the composed class.	Ragonis and Ben-Ari (2005a)
99	ObjState	During a method call, an object attribute is duplicated as a local variable. Assignments update both.	Sajaniemi et al. (2008)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
100	ObjState	During a method call, an object attribute is duplicated as a local variable. The local variable is initialized from the object, updated by the method, then returned to object at end.	Sajaniemi et al. (2008)
101	ObjState	During a method call, an object attributes is duplicated as a local variable, which is initialized to a default value (e.g., zero). Assignments only affect the local variable, not the object.	Sajaniemi et al. (2008)
102	ObjState	Parameters belong to the called object.	Sajaniemi et al. (2008)
103	ObjState	Local variables belong to the called object.	Sajaniemi et al. (2008)
104	ObjState	A main method's local variables belong to the called object.	Sajaniemi et al. (2008)
105	ObjState	An object can only hold instance variables of a single type.	Holland et al. (1997)
106	ObjState	No object attributes within objects; they only exist locally during method calls.	Sajaniemi et al. (2008)
107	ObjState	An object cannot be the value of an attribute.	Ragonis and Ben-Ari (2005a)
108	ObjState	Attributes of composed classes include object attributes from the simple classes instead of the objects.	Ragonis and Ben-Ari (2005a)
109	ObjState	Attributes of composed classes include object attributes from the simple classes in addition to the objects.	Ragonis and Ben-Ari (2005a)
110	ObjState	An object is a wrapper for a single variable. The object is equated with the variable.	Holland et al. (1997); Vainio (2006)
111	ObjState	Attributes in a simple class are automatically replicated in the composed class by transferring its meaning.	Ragonis and Ben-Ari (2005a)
112	ObjState	To change the value of an attribute of an object of a simple class that is the value of an attribute in an object of a composed class, you need to construct a new object.	Ragonis and Ben-Ari (2005a)
113	ObjState	Difficulty grasping what the properties are that represent an object's state.	Sorva (2007)
114	ObjState	The name of the variable that the object was most recently assigned to is a part of an object's state.	Sorva (2007, 2008)
115	ObjState	Storing an object in memory means storing a copy of the program code of the object's class.	Sorva (2007)
116	ObjState	Storing an object means storing the constructor parameters given upon object creation. These parameter values unambiguously define the object. (Methods do not affect state.).	Sorva (2007)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
117	ObjState	You can define a (Java) method that adds an attribute to the class.	Ragonis and Ben-Ari (2005a)
118	ObjState	Objects of a simple class, used as values of the attributes of a composed class, have to be identical.	Ragonis and Ben-Ari (2005a)
119	ObjState	In a composed (Java) class, you can develop a method that adds an attribute of a simple class to the composed class.	Ragonis and Ben-Ari (2005a)
120	ObjState	In a composed (Java) class, you can develop a method that removes an attribute of a simple class from the composed class.	Ragonis and Ben-Ari (2005a)
121	ObjState	Objects of the same class cannot have equal attribute values.	Ragonis and Ben-Ari (2005a)
122	ObjState	Attributes of the simple class must be directly accessed from the composed class instead of through an interface.	Ragonis and Ben-Ari (2005a)
123	ObjState	Objects are allocated the same amount of memory regardless of definition and instantiation.	Kaczmarczyk et al. (2010)
124	Methods	Objects 'know' which methods are operating on them (rather than method calls 'knowing' which object they operate on).	Sajaniemi et al. (2008)
125	Methods	Cannot have methods with the same name in different classes.	Fleury (2000); Ragonis and Ben-Ari (2005a)
126	Methods	The dot operator can only be applied to methods.	Fleury (2000)
127	Methods	You can define a method that replaces the object itself.	Ragonis and Ben-Ari (2005a)
128	Methods	You can define a method that destroys the object itself.	Ragonis and Ben-Ari (2005a)
129	Methods	You can define a method that divides the object into two different objects.	Ragonis and Ben-Ari (2005a)
130	Methods	Methods can only do assignment.	Holland et al. (1997)
131	Methods	Methods from the simple class are not used; instead, new equivalent methods are defined and duplicated in the composed class.	Ragonis and Ben-Ari (2005a)
132	Methods	You can invoke a method on an object only once.	Ragonis and Ben-Ari (2005a)
133	Methods	Difficulties in understanding that a method can be invoked on any object of the class.	Ragonis and Ben-Ari (2005a)
134	Methods	Methods can only be invoked on objects of the composed class, not on objects of the simple class defined as values in its attributes.	Ragonis and Ben-Ari (2005a)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
135	Methods	After a composed class is defined, new methods cannot be defined in the simple class.	Ragonis and Ben-Ari (2005a)
136	Methods	A (Java) method must always be invoked on an explicit object.	Ragonis and Ben-Ari (2005a); Hristova et al. (2003)
137	Methods	Static and dynamic existence of methods mixed up.	Sajaniemi et al. (2008)
138	Methods	Methods that are declared in a simple class have to be declared again in a composed class for each of the simple objects.	Ragonis and Ben-Ari (2005a)
139	OtherOOP	There is no need for mutators and accessors for attributes that are of a simple class within a composed class.	Ragonis and Ben-Ari (2005a)
140	OtherOOP	Object lookup involves searching through each object in memory for a suitable ID.	local
141	OtherOOP	Objects are stored in folders on the hard drive and lookup means searching through them.	local
142	OtherOOP	An object is just a piece of code (and not a dynamic actor at runtime).	Eckerdal and Thuné (2005)
143	OtherOOP	An object is just a record.	Holland et al. (1997)
144	OtherOOP	An object is a 'way of working' consisting of action: expressions, return expressions etc. It has a control flow. No concept of state.	Vainio (2006)
145	OtherOOP	Difficulties in understanding how one class recognizes another.	Ragonis and Ben-Ari (2005a)
146	OtherOOP	Difficulties in understanding inheriting functionality (methods).	Détienne (1997)
147	OtherOOP	Difficulties in understanding how the computer knows what class attributes and methods are.	Ragonis and Ben-Ari (2005a)
148	OtherOOP	Assignment confused with subclassing.	Ma (2007)
149	OtherOOP	Inheritance hierarchies express the parts of composite classes.	Détienne (1997)
150	Misc	Difficulties understanding the effect of input function calls on execution.	Bayman and Mayer (1983); du Boulay (1986); Putnam et al. (1986)
151	Misc	Confusion between an array and its cell.	du Boulay (1986)
152	Misc	Difficulties with dealing with 2D array subscripts and dimensions.	du Boulay (1986)

Continues on next page

Table A.1 continued

No.	Topic	Description	Source
153	Misc	Difficulties with arrays containing indices as data.	du Boulay (1986)
154	Misc	Values of conditional expressions get printed out.	Putnam et al. (1986)
155	Misc	Numbers are just numbers. (Why have `int` and `float` separately?)	Hristova et al. (2003)
156	Misc	A type is a set of constraints on values.	Vainio (2006)
157	Misc	Types can change on the fly (in Java).	Vainio (2006)
158	Misc	Confusion between data in memory and data on screen.	Bayman and Mayer (1983)
159	Misc	The computer keeps what has been printed in memory (as part of state?).	Bayman and Mayer (1983)
160	Misc	Confusing textual representations with each other, e.g., the string "456" with the number.	du Boulay (1986)
161	Misc	Boolean values are just something used in conditionals and not data comparable to numbers or strings.	local
162	Misc	A `for` loop control variable constrains the values that can be input within the loop.	Putnam et al. (1986); Sleeman et al. (1986)

Appendix B

Example Programs from Course Assignments

Below is the program code used within UUhistle in the program-reading assignments of the CS1–Imp–Pyth course in spring 2010 (see Section 16.4). This includes both animations and visual program simulation exercises.

The identifiers and string literals used below roughly correspond to the original Finnish ones seen by the students.

Assignment 1.1 (animation)

```
marks = 200 + 250
euros = marks / 5.94573
```

Assignment 1.2 (VPS)

```
celsius = 100
fahrenheit = 1.8 * celsius + 32
```

Assignment 1.3 (VPS)

```
first = 10
second = -20
temp = first
first = second
second = temp
```

Assignment 1.4 (VPS)

```
first = 3
first = first + 1
first = 1 + first
first = first + first
```

Assignment 1.5 (animation)

```
name = raw_input('What is your name?')
print 'Delightful name,', name
```

Assignment 1.6 (VPS)

```
celsius = raw_input('Enter temperature in celsius:')
fahrenheit = 1.8 * float(celsius) + 32
print fahrenheit
```

Assignment 1.7 (VPS)

```
marks = float(raw_input('Enter a price in Finnish marks:'))
print 'That is', marks / 5.94573, 'euros.'
```

Assignment 2.1 (animation)

```
line = raw_input('Enter a number:')
number = float(line)
if number > 5000:
    print 'And I should have seen the one that got away?'
else:
    print 'That is all you got?'
print 'The End.'
```

Assignment 2.2 (VPS)

```
denominator = 4
if denominator == 0:
    print 'Chuck Norris divides by zero, you do not.'
else:
    print 1000 / denominator
```

Assignment 2.3 (VPS)

```
age = 25
if age >= 0:
    adult = age >= 17
    if adult:
        print 'Adult: ', age - 18
    else:
        print 'Yet a child: ', 18 - age
    print adult
else:
    print 'One for the future.'
print vuodet
```

Assignment 3.1 (animation)

```
HOW_MANY = 3
i = 0
sum = 0
while i < HOW_MANY:
    line = raw_input('Enter measurement:')
    sum = sum + float(line)
    i = i + 1
print 'Average:', sum / HOW_MANY
```

Assignment 3.2 (VPS)

```
i = 0
while i < 7:
    i = i + 2
    print i
    i = i + 1
print i
```

Assignment 4.1 (animation)

```
def greet(name, magic_number):
    print 'Hi,', name
    return magic_number + 1

result = greet('Esa the Engineer', 7)
print result * 5
print greet('Metal-Ari', 666)
```

Assignment 4.2 (VPS)

```
def get_euros():
    line = raw_input('Enter a price in euros:')
    return row

input = get_euros()
print float(input)
```

Assignment 4.3 (VPS)

```
def calculate(first, second):
    return second * 2 + first

result = calculate(3, 2)
result = calculate(result, result + 1)
```

Assignment 4.4 (VPS)

```
def calculate(first, second):
    intermediate = first - second
    return intermediate * intermediate

def main():
    number1 = int(raw_input('Enter an integer:'))
    number2 = 10
    intermediate = calculate(number1, number2)
    print intermediate + 2

main()
```

Assignment 5.1 (animation)

```
def check_info(name, age):
    return len(name) >= 0 and age >= 0

def print_info(name, age):
    if not check_info(name, age):
        return False
    print 'First name:', name
    print 'Age (years):', age
    return True

if print_info('Johan', 15):
    print print_info('Peewit', -1000)
```

```
else:
    print 'Better luck next time.'
```

Assignment 5.2 (animation)

```
def calculate(first, second):
    return second * 3 + first

def double(number):
    return number * 2

print double(double(5))
print calculate(double(3), 2 + calculate(5, 1))
```

Assignment 8.1 (animation)

```
def count_sum(sum, left):
    if left > 0:
         newest = float(raw_input('Enter a number:'))
         return count_sum(sum + newest, left - 1)
    else:
         return sum

print 'The sum of the numbers is:', count_sum(0, 3)
```

Assignment 8.2 (VPS)

```
# Determines the factorial of a positive integer.
def fact(n):
    if n < 3:
        return n
    else:
        return fact(n-1) * n

print 'result:', fact(5)
```

Assignment 9.1 (animation)

```
ride = Car(50)
ride.refuel(40)
added = ride.refuel(60)
print added
ride.drive(15)
print ride.get_fuel()

second_car = Car(60)
second_car.refuel(10)
```

The execution of the above code was animated. In addition, the following class was built into the assignment as 'hidden code' (see Section 13.3), and worked as a black box.

```
class Car:
  def init(self, tank_capacity):
      self.__tank_capacity = tank_capacity
      self.__gas = 0
```

```
  def refuel(self, liters):
      actually_added = min(liters, self.__tank_capacity - self.__gas)
      self.__gas = self.__gas + actually_added
      return actually_added

  def drive(self, consumption):
      if self.__gas < consumption:
          return False
      self.__gas = self.__gas - consumption
      return True

  def get_fuel(self):
      return self.__gas
```

Assignment 9.2 (VPS)

```
ride = Car(45)
ride.refuel(15)
wheels = Car(60)
work_car = ride
work_car.refuel(20)
print ride.get_fuel()
print work_car.get_fuel()
work_car = wheels
print work_car.get_fuel()
```

This VPS assignment used the same (hidden) car class as the previous animation.

Assignment 9.3 (animation)

```
class Car:
  def __init__(self, tank_capacity):
      self.__tank_capacity = tank_capacity
      self.__gas = 0

  def refuel(self, liters):
      actually_added = min(liters, self.__tank_capacity - self.__gas)
      self.__gas = self.__gas + actually_added
      return actually_added

  def drive(self, consumption):
      if self.__gas < consumption:
          return False
      self.__gas = self.__gas - consumption
      return True

  def get_fuel(self):
      return self.__gas

ride = Car(50)
ride.refuel(40)
added = ride.refuel(60)
print added
```

```
ride.drive(15)
print ride.get_fuel()

second_car = Car(60)
second_car.refuel(10)
```

Assignment 9.4 (VPS)

```
class Person:
    def __init__(self, first_name, profession):
        self.__name = first_name
        self.__profession = profession

    def greet(self, encounteree):
        return self.__name + '. Pleased to meet you, ' + encounteree.__name

    def get_profession(self):
        return self.__profession

first = Person('Babar', 'doctor')
print first.get_profession()
second = Person('Sapphira', 'biologist')
print first.greet(second)
```

Appendix C

Example Programs from Experiment

These programs were used in the experiment from Chapter 19.

The identifiers and string literals used below roughly correspond to the original Finnish ones seen by the students.

Initial example

This code was shown to the treatment group as a UUhistle v0.2 animation and explained to the control group textually, line by line.

```
first = [ 10, 5, 10, 6 ]
print first[3]
second = [ 3, 1, -2 ]
print second
second[2] = second[2] + 1
print second[2]
```

First VPS task

This code was given to the treatment group as a VPS exercise. The control group was also provided the correct output and asked to figure out and explain why the program worked as it did.

```
first = [ 10, 5, 0 ]
first[1] = -5
value = first[2]
print first
second = [ 3, 1, 3, value ]
print second[value]
third = second
second[3] = 100
print second
print third
second = third
print second
```

Second VPS task

```
def weird(list, index):
    test = list
    list = [3, list[0]]
    test[index] = list[0]
    return test[1]
```

```
def quaint(list, element):
    list[0] = element
    return list

list = ['cobra', 'python', 'mamba']
print weird(list, 1)
second = quaint(['sheep', 'chamois', 'sheep'], 'grandma')
third = quaint(second, 'wolf')
print list
print second
print third
```

Q1 (pretest and post-test)

The pre- and post-test questions required the students to produce the correct output of the program.

```
a = [ 4, 3, 6, 1, 4 ]
print a[1]
number = 2
b = [ 1, 3, number ]
print b[2]
b[2] = 10
c = a
c[number] = 20
print c
print b
print a
```

Q2 (pretest and post-test)

```
def inspect(a):
    print a[0]

def lookup(a, b):
    print a[b]
    return 1

a = [ 2, 3, 4, 3 ]
b = [ 3, 7, 1 ]
inspect(a)
lookup(a, 1)
lookup(b, lookup(a, b[0]))
```

Q3 (pretest and post-test)

```
def heigh_ho(list, another):
    temp = list
    list = another
    another = temp
    return another

a = [ "off", "to", "work" ]
b = [ "we", "dig", "dig", "dig", "dig" ]
print heigh_ho(a, b)
```

Q4 (pretest and post-test)

```
def tralala(list, first, second):
    temp = list[first]
    list[first] = list[second]
    list[second] = temp
    return list

a = [ "cat", "dog", 343, "giraffe" ]
tralala(a, 2, 0)
b = a
c = tralala(b, 1, 2)
print a
print b
print c
a = tralala(a, 1, 2)
a[0] = "santa claus"
print a
print b
print c
```

Appendix D

3×10 Bullet Points for the CS1 Teacher

Why should I use VPS in my introductory programming course?

- VPS can help students learn to read program code.
- VPS addresses something important: what happens behind the scenes as programs get executed. That is, the hidden stuff that causes problems for novices (perhaps especially in OOP).
- VPS makes what is often left tacit or glossed over not only visible but manipulable.
- VPS can help prevent misconceptions about programming concepts and constructs, and to improve on existing ones.
- Students who fail to develop a dynamic perspective on programs and the 'notional machine' that runs them can get badly stuck, unable to learn about further concepts.
- VPS is a cognitively engaging form of visualization use; the learner can't just 'press play' and switch off.
- VPS can illustrate to novices what to keep track of when tracing programs, and can serve as a stepping stone towards the use of regular debuggers.
- Tool-supported VPS provides a form of deliberate practice that can help a principled mental model of program execution become ingrained.
- A VPS system can automatically assess students' solutions and give automatic feedback, saving staff resources.
- A tool-supported VPS exercise only takes a few minutes of student time, and even a bunch of them doesn't take *that* much time.

I'm not convinced...

- You learn to program by programming! *Answer:* I agree that you don't learn to program without programming. However, other learning activities can help, too. In fact, there is plenty of evidence that shows novices don't learn to solve problems best by only solving problems. (See Section 4.5.) Studying existing programs is also useful.
- Too much hand-holding. Many people just don't have what it takes to program. You're perpetuating your storm-in-a-teacup field of research. *Answer:* VPS and this thesis are indeed useless for the purpose of building a panning-for-gold CS1 that seeks to get rid of the ones that "aren't good enough to be programmers". Lister (2011a) has recently discussed how we might transcend the old "Is programming an innate ability?" debate through a neo-Piagetian perspective, which recognizes that the ability to reason in abstract conceptual terms about programs comes spontaneously to some people, but that many others can also be taught this (domain-specific) form of reasoning.

Already now, and especially in the future, a lot of people will need to learn at least the basics of programming. Some hand-holding is needed.[1]

- The basics are really not that hard. The students are just not putting in the effort, that's why they struggle with CS1. *Answer:* That is a problem in some cases (and good pedagogy can help with that, too!), but there definitely are many CS1 students at many institutions who are working hard but failing (see, e.g., Mason and Cooper, 2012).

- My students are already overloaded with content. I can't fit in the 'notional machine', too. *Answer:* It's already in. Your students deal with a notional machine, anyway – it is implied by the programming language whether it is made explicit in teaching or not. VPS makes the machine tangible.

- VPS doesn't address the big problem of CS1: even when students get how the individual statements work, they don't know how to put them together. *Answer:* That is one of the big problems of CS1. There are other significant ones. (See, e.g., Section 5.6.) VPS doesn't directly address that problem, but practice with lower-level issues can alleviate the cognitive load inherent in program authoring.

- VPS is a really artificial activity. Also, students just reproduce what the system wants them to 'know'. To develop authentic professional skills, we need authentic activities. *Answer:* Authenticity is important, too, but not all useful learning requires it. Cognitive overload from authentic complexity is a big concern in what comes to beginners. We need to find a balance through a combination of learning activities. (See, e.g., Section 10.1.)

- The visualization and the VPS controls are too much work for students to learn. *Answer:* There is an overhead, and it is true that he usability and cognitive demands of VPS interfaces must be further investigated. Embedding a single VPS assignment in a course is probably not worth it. Still, in our studies, many students had no great trouble with the GUI and the usability of our proof-of-concept prototype received generally positive reviews.

- It's easier to just stick to explaining how programs work and/or having students view program animations. *Answer:* Those techniques can also be useful, but have a potential weakness in learner engagement. Future research can tell us more about the relationship between program animation and VPS, but the overall message from educational software visualization research is that it is a good idea to require students to actively do things with visualizations. In our experiment (in Chapter 19), for certain content, VPS was more effective than having students read explanations and analyze code without a visualization. Future VPS systems may be increasingly efficient in giving personalized feedback and directly addressing students' misconceptions in ways that less interactive presentations cannot (cf. Section 15.3).

- I prefer to have my students draw program state diagrams themselves and/or role-play program execution in the classroom. *Answer:* Those techniques can also be useful, but are somewhat cumbersome. Tool-supported VPS enables convenient practice on many programs at the learner's own pace, and automatic assessment and feedback. It is compatible with large-class scenarios and distance education.

- You sound awfully positive, considering your thesis has demonstrated that many students didn't get VPS and used mechanical trial and error to get past assignments. *Answer:* Many did get it and did learn from it, too. VPS encourages cognitive engagement with a visualization, but doesn't guarantee engagement. It can be useful, even if it isn't useful all the time and for everyone. Our research has also suggested how to improve the impact of VPS...

[1] I hope nobody interprets me as saying that VPS is meant for the untalented. There is evidence within this very book that it is possible for smart and experienced students, too, to learn from VPS.

Okay, what should I take into consideration as I adopt VPS?

- Don't assume students will just get the point of VPS on their own. A good introduction to the visualization and VPS is essential. First impressions matter. It is important that students get the idea early that VPS can help them. A learning tool is only as good as what the learner makes of it.
- Integrate the visualization into the rest of the learning and teaching environment. Use program animation in class to explore puzzling questions on program behavior (e.g., to explain new constructs and to explore bugs). Use the visualization to demonstrate the non-viability of common misconceptions. Link animations and VPS assignments to other assignments and make explicit how they are connected (e.g., a VPS assignment prepares for a program-writing assignment).
- Use the visualization and VPS throughout CS1.
- Example selection is key. Program examples must illustrate how underlying principles – of evaluation order, reference semantics, etc. – make a real difference to program behavior. Misconception catalogues (e.g., Appendix A) can inspire the design of examples.
- Students will learn about what they think about. They think about what the visualization system makes relevant for them. Carefully consider the learning goals of each VPS assignment and the expected user interactions within the VPS system, and make sure they are well aligned. Use hybrid forms of program visualization: VPS combined with popup dialogs, multiple-choice questions, program animation – anything to draw students' attention to the important things.
- Instructions and program animations help students learn to do VPS, but students often don't pay much attention to them. Provide incentives to pay attention. If possible, have students work on VPS during supervised sessions or provide initial training (with personal feedback from tutor) in how to make the most of VPS. Make sure you don't only provide instructions and feedback on the operational level; also address the underlying concepts.
- Listen to how your students reason as they do VPS. Engage them in dialogue. Guide them towards the intended object of learning. Have students explain program behavior to themselves, to you, or to a partner. When students get stuck, encourage them to focus not on GUI operations but on what the next step in the program's execution is. Discourage superficial VPSing tactics by showing how conceptual reasoning leads to the right answers.
- Analyze students' mistakes. VPS can help discover how students think. Encourage students to analyze their own mistakes, too.
- Be sensitive to the weaknesses of VPS. Some students rely on trial and error tactics or guessing based on superficial resemblances between code and graphical elements. Some students don't associate the visualization with computer behavior, and some don't see the connection between computer behavior as presented in VPS and authentic programming skills. Tedium and a lack of focus on key concepts are genuine threats to the success of VPS that must be fought through careful assignment and system design. Accept that not everybody will enjoy VPS.
- Explore the variation in your students. Some of them might benefit more from VPS than others (who have more programming experience, perhaps, and less trouble with program dynamics). VPS could be targeted at only some of the students while others work on other assignments.

Appendix E

Statement of the Author's Contribution

- The review of program visualization systems that appears in Chapter 11 was conducted by the author of the thesis in collaboration with Ville Karavirta and Lauri Malmi. The author had a leading role in this work.
- The software system introduced in Chapter 13 is joint work by the author of the thesis and Teemu Sirkiä as described at the beginning of the chapter. Parts of Chapters 13 and 15 are based on joint publications with Teemu Sirkiä, in which the author of the thesis was the primary author.
- Part V describes empirical studies which were conducted by the the author of the thesis in collaboration with Lauri Malmi, who participated in research design and the analysis of data. Kimmo Kiiski and Teemu Koskinen aided with data collection, as did Jan Lönnberg, who also participated in the analysis described in Chapter 17. Some more details on the researcher roles in each study can be found in the corresponding chapters. The author of the thesis had a leading role in all the empirical work described.
- Except as noted above, the work is that of the author. He has written all the chapters of this book.

References

ACM and IEEE Computer Society (2001). Computing Curricula 2001: Computer Science.
URL http://www.acm.org/education/curric_vols/cc2001.pdf

ACM and IEEE Computer Society (2008). Computer Science Curriculum 2008: An Interim Revision of CS 2001.
URL http://www.acm.org/education/curricula/ComputerScience2008.pdf

Adelson, B. and Soloway, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, 11(11):1351–1360.

Ahoniemi, T. and Lahtinen, E. (2007). Visualizations in Preparing for Programming Exercise Sessions. *Electronic Notes in Theoretical Computer Science*, 178:137–144.

Airila, M. and Pekkanen, M. A. (2002). *Tekniikan alan väitöskirjaopas*. Helsinki University of Technology. A doctoral dissertation guide for technical fields of research, in Finnish.

Åkerlind, G. S. (2005a). Learning about Phenomenography: Interviewing, Data Analysis and the Qualitative Research Paradigm. In: J. A. Bowden and P. Green (eds.), *Doing Developmental Phenomenography*, pp. 63–73. RMIT University Press.

Åkerlind, G. S. (2005b). Phenomenographic Methods: A Case Illustration. In: J. A. Bowden and P. Green (eds.), *Doing Developmental Phenomenography*, pp. 103–127. RMIT University Press.

Åkerlind, G. S. (2005c). Variation and Commonality in Phenomenographic Research Methods. *Higher Education Research & Development*, 24(4):321–334.

Ala-Mutka, K. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102.

Alaoutinen, S. and Smolander, K. (2010). Student Self-Assessment in a Programming Course Using Bloom's Revised Taxonomy. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pp. 155–159. ACM.

Algoviz (n.d.). Algoviz.org: The Algoritm Visualization Portal.
URL http://www.algoviz.org/

Allen, E., Cartwright, R., and Stoler, B. (2002). DrJava: A Lightweight Pedagogic Environment for Java. *SIGCSE Bulletin*, 34(1):137–141.

Alphonce, C. and Ventura, P. (2002). Object Orientation in CS1-CS2 by Design. *SIGCSE Bulletin*, 34(3):70–74.

Ambrósio, A. P. L. and Costa, F. M. (2010). Evaluating the Impact of PBL and Tablet PCs in an Algorithms and Computer Programming Course. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 495–499. ACM.

Anderson, J. R. (2009). *Cognitive Psychology and its Implications*. Worth Publishers, 7th edition.

Anderson, J. R., Conrad, F. G., and Corbett, A. T. (1989). Skill Acquisition and the LISP Tutor. *Cognitive Science*, 13(4):467–505.

Anderson, J. R., Farrell, R., and Sauers, R. (1984). Learning to Program in LISP. *Cognitive Science: A Multidisciplinary Journal*, 8(2):87–129.

Anderson, J. R., Fincham, J. M., and Douglass, S. (1997). The Role of Examples and Rules in the Acquisition of a Cognitive Skill. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(4):932–945.

Anderson, J. R., Greeno, J. G., Reder, L. M., and Simon, H. A. (2000a). Perspectives on Learning, Thinking, and Activity. *Educational Researcher*, 29(4):11–13.

Anderson, J. R., Pirolli, P., and Farrell, R. (1988). Learning to Program Recursive Functions. In: M. T. H. Chi, R. Glaser, and M. J. Farr (eds.), *The Nature of Expertise*, pp. 153–183. Lawrence Erlbaum.

Anderson, J. R., Reder, L. M., and Lebiere, C. (1996). Working Memory: Activation Limitations on Retrieval. *Cognitive Psychology*, 30(3):221–256.

Anderson, J. R., Reder, L. M., and Simon, H. A. (2000b). Applications and Misapplications of Cognitive Psychology to Mathematics Education. *Texas Educational Review*, 1(2):29–49.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J., and Wittrock, M. C. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman.

Anderson, R. C. (1977). The Notion of Schemata and the Educational Enterprise: General Discussion of the Conference. In: R. C. Anderson, R. J. Spiro, and W. E. Montague (eds.), *Schooling and the Acquisition of Knowledge*, pp. 415–431. Lawrence Erlbaum.

Andrianoff, S. K. and Levine, D. B. (2002). Role Playing in an Object-Oriented World. *SIGCSE Bulletin*, 34(1):121–125.

Atherton, J. S. (n.d.). Learning and Teaching: SOLO Taxonomy. Accessed October 2011. URL `http://www.learningandteaching.info/learning/solo.htm`

Atkinson, R. C. and Shiffrin, R. M. (1968). Human Memory: A Proposed System and Its Control Processes. In: K. W. Spence and J. T. Spence (eds.), *The Psychology of Learning and Motivation: Advances in Research and Theory*, volume 2, pp. 89–195. Academic Press.

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000). Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research*, 70(2):181–214.

Auguston, M. and Reinfelds, J. (1994). A Visual Miranda Machine. In: *Software Education Conference, 1994: Proceedings*, pp. 198–203. IEEE.

Ausubel, D. P. (1968). *Educational Psychology: A Cognitive View*. Holt, Rinehart and Winston.

Auvinen, T. (2009). *Rubyric – A Rubrics-Based Online Assessment Tool for Effortless Authoring of Personalized Feedback*. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology.

Babbage, C. (1864). *Passages from the Life of a Philosopher*. Longman Green.

Baddeley, A. D. (2000). The Episodic Buffer: A New Component of Working Memory? *Trends in Cognitive Sciences*, 4(11):417–423.

Baddeley, A. D. and Hitch, G. J. (1974). Working Memory. *Recent Advances in Learning and Motivation*, 8:47–86.

Bareiss, R. and Radley, M. (2010). Coaching via Cognitive Apprenticeship. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 162–166. ACM.

Bares, W. H., Zettlemoyer, L. S., and Lester, J. C. (1998). Habitable 3D Learning Environments for Situated Learning. In: B. P. Goettl, H. M. Halff, C. L. Redfield, and V. J. Shute (eds.), *Proceedings of the 4th International Conference on Intelligent Tutoring Systems*, volume 1452 of *Lecture Notes in Computer Science*, pp. 76–85. Springer.

Barker, J. (2000). *Beginning Java Objects: From Concepts to Code*. Apress.

Barnes, D. J. and Kölling, M. (2006). *Objects First with Java – A Practical Introduction using BlueJ*. Prentice-Hall / Pearson Education, 3rd edition.

Bartlett, F. C. (1932). *Remembering: A Study in Experimental and Social Psychology*. Cambridge University Press.

Bataller Mascarell, J. (2011). Visual Help to Learn Programming. *ACM Inroads*, 2(4):42–48.

Bayman, P. and Mayer, R. E. (1983). A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Communications of the ACM*, 26(9):677–679.

BBC Sport (2004). All in the Mind. In: *BBC Sport Football Website*. British Broadcasting Corporation. URL `http://news.bbc.co.uk/sport2/hi/football/eng_prem/3758916.stm`

Beck, K. and Cunningham, W. (1989). A Laboratory for Teaching Object Oriented Thinking. *SIGPLAN Notices*, 24(10):1–6.

Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M. (2006). Analyzing Individual Differences in Program Comprehension. *Technology, Instruction, Cognition and Learning*, 3(3):205–232.

Ben-Ari, M. (2001a). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73.

Ben-Ari, M. (2001b). Program Visualization in Theory and Practice. *Informatik / Informatique, Special Issue on Visualization of Software*, pp. 8–11.

Ben-Ari, M. (2004). Situated Learning in Computer Science Education. *Computer Science Education*, 14(2):85–100.

Ben-Ari, M. (2005). Situated Learning in 'This High-Technology World'. *Science & Education*, 14(3):367–376.

Ben-Ari, M. (2010). Objects Never? Well, Hardly Ever! *Communications of the ACM*, 53(9):32–35.

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N., and Sutinen, E. (2011). A Decade of Research and Development on Program Animation: The Jeliot Experience. *Journal of Visual Languages & Computing*, 22:375–384.

Ben-Ari, M. and Yeshno, T. (2006). Conceptual Models of Software Artifacts. *Interacting with Computers*, 18(6):1336–1350.

Ben-Bassat Levy, R. and Ben-Ari, M. (2007). We Work So Hard and They Don't Use It: Acceptance of Software Tools by Teachers. *SIGCSE Bulletin*, 39(3):246–250.

Ben-Bassat Levy, R., Ben-Ari, M., and Uronen, P. A. (2003). The Jeliot 2000 Program Animation System. *Computers & Education*, 40(1):1–15.

Bennedsen, J. and Caspersen, M. E. (2008). Exposing the Programming Process. In: J. Bennedsen, M. E. Caspersen, and M. Kölling (eds.), *Reflections on the Teaching of Programming: Methods and Implementations*, pp. 6–16. Springer.

Bennedsen, J. and Schulte, C. (2006). A Competence Model for Object Interaction in Introductory Programming. In: P. Romero, J. Good, E. Acosta-Chaparro, and S. Bryant (eds.), *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group*, PPIG'06, pp. 215–229. PPIG.

Bennedsen, J. and Schulte, C. (2007). What Does "Objects-First" Mean? An International Study of Teachers' Perceptions of Objects-First. In: R. Lister and Simon (eds.), *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 21–29. Australian Computer Society.

Bennedsen, J. and Schulte, C. (2010). BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs? *ACM Transactions on Computing Education*, 10(2):1–22.

Bergantz, D. and Hassell, J. (1991). Information Relationships in PROLOG Programs: How Do Programmers Comprehend Functionality? *International Journal of Man–Machine Studies*, 35(3):313–328.

Bergin, J. (2000). Why Procedural is the Wrong First Paradigm if OOP is the Goal. Accessed October 2011.
URL `http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html`

Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. (2005). *Karel J Robot: A Gentle Introduction to The Art of Object-Oriented Programming in Java*. Dream Songs Press.

Berglund, A. (2002). *On the Understanding of Computer Network Protocols*. Licentiate's thesis, Department of Information Technology, Uppsala University.

Berglund, A. (2005). *Learning Computer Systems in a Distributed Project Course: The What, Why, How and Where*. Doctoral dissertation, Department of Information Technology, Uppsala University.

Berglund, A. and Lister, R. (2007). Debating the OO Debate: Where is the Problem? In: R. Lister and Simon (eds.), *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 171–174. Australian Computer Society.

Berglund, A. and Lister, R. (2010). Introductory Programming and the Didactic Triangle. In: T. Clear and J. Hamer (eds.), *Twelfth Australasian Computing Education Conference (ACE 2010)*, volume 103 of *CRPIT*, pp. 35–44. Australian Computer Society.

Besnard, D., Greathead, D., and Baxter, G. (2004). When Mental Models Go Wrong: Co-Occurrences in Dynamic, Critical Systems. *International Journal of Human-Computer Studies*, 60(1):117–128.

Bhuiyan, S. H., Greer, J. E., and McCalla, G. I. (1990). Mental Models of Recursion and Their Use in the SCENT Programming Advisor. In: *Proceedings of the International Conference on Knowledge Based Computer Systems*, KBCS '89, pp. 135–144. Springer.

Bickhard, M. H. (1995). World Mirroring versus World Making: There's Gotta be a Better Way. In: L. P. Steffe and J. E. Gale (eds.), *Constructivism in Education*, pp. 229–267. Lawrence Erlbaum.

Bielaczyc, K., Pirolli, P. L., and Brown, A. L. (1995). Training in Self-Explanation and Self-Regulation Strategies: Investigating the Effects of Knowledge Acquisition Activities on Problem Solving. *Cognition and Instruction*, 13(2):221–252.

Biermann, A. W., Fahmy, A. F., Guinn, C., Pennock, D., Ramm, D., and Wu, P. (1994). Teaching a Hierarchical Model of Computation with Animation Software in the First Course. *SIGCSE Bulletin*, 26(1):295–299.

Biesta, G. J. J. and Burbules, N. C. (2003). *Pragmatism and Educational Research*. Rowman & Littlefield.

Biggs, J. and Tang, C. (2007). *Teaching for Quality Learning at University*. McGraw-Hill, 3rd edition.

Biggs, J. B. and Collis, K. F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Birch, M. R., Boroni, C. M., Goosey, F. W., Patton, S. D., Poole, D. K., Pratt, C. M., and Ross, R. J. (1995). DYNALAB: A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation. *SIGCSE Bulletin*, 27(1):29–33.

Blackwell, A. F. (2000). Dealing with New Cognitive Dimensions. In: *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community*. University of Hertfordshire.

Bloom, B. S. (1956). *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley.

Boisvert, C. R. (2009). A Visualisation Tool for the Programming Process. *SIGCSE Bulletin*, 41(3):328–332.

Bonar, J. and Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human–Computer Interaction*, 1(2):133–161.

Booth, J. (2006). On the Mastery of Philosophical Concepts: Socratic Discourse and the Unexpected 'Affect'. In: J. H. F. Meyer and R. Land (eds.), *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, pp. 173–181. Routledge.

Booth, S. (1992). *Learning to Program: A Phenomenographic Perspective*. Doctoral dissertation, University of Gothenburg.

Booth, S. (2001a). Learning Computer Science and Engineering in Context. *Computer Science Education*, 11(3):169–188.

Booth, S. (2001b). Learning to Program as Entering the Datalogical Culture: A Phenomenographic Exploration. In: *9th European Conference for Research on Learning and Instruction*, EARLI '01.

Bornat, R., Dehnadi, S., and Simon (2008). Mental Models, Consistency and Programming Aptitude. In: *Proceedings of the Tenth Conference on Australasian Computing Education*, ACE '08, pp. 53–61. Australian Computer Society.

Boroni, C. M., Eneboe, T. J., Goosey, F. W., Ross, J. A., and Ross, R. J. (1996). Dancing with DynaLab: Endearing the Science of Computing to Students. *SIGCSE Bulletin*, 28(1):135–139.

Boustedt, J. (2009). Students' Understanding of the Concept of Interface in a Situated Context. *Computer Science Education*, 19(1):15–36.

Boustedt, J. (2012). Students' Different Understandings of Class Diagrams. *Computer Science Education*, 22(1):29–62.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2007). Threshold Concepts in Computer Science: Do They Exist and Are They Useful? *SIGCSE Bulletin*, 39(1):504–508.

Bowden, J. (2005). Reflections on the Phenomenographic Team Research Process. In: J. A. Bowden and P. Green (eds.), *Doing Developmental Phenomenography*, pp. 11–31. RMIT University Press.

Bowden, J. and Marton, F. (2004). *The University of Learning*. Routledge.

Bowden, J. A. (2000). The Nature of Phenomenographic Research. In: J. A. Bowden and E. Walsh (eds.), *Phenomenography*, pp. 1–18. RMIT University Press.

Bowden, J. A. and Green, P. (eds.) (2005). *Doing Developmental Phenomenography*. RMIT University Press.

Bower, M. (2008). A Taxonomy of Task Types in Computing. *SIGCSE Bulletin*, 40(3):281–285.

Brabrand, C. and Dahl, B. (2009). Using the SOLO Taxonomy to Analyze Competence Progression of University Science Curricula. *Higher Education*, 58(4):531–549.

Breakwell, G., Hammond, S., Fife-Schaw, C., and Smith, J. A. (eds.) (2006). *Research Methods in Psychology*. Sage Publications.

Brewer, W. F. (1987). Schemas versus Mental Models in Human Memory. In: P. Morris (ed.), *Modelling Cognition*, pp. 187–197. John Wiley & Sons.

Brewer, W. F. (2002). Learning Theory: Schema Theory. Accessed March 2010.
URL `http://www.answers.com/topic/learning-theory-schema-theory`

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man–Machine Studies*, 18(6):543–554.

Brown, M. H. (1988). Exploring Algorithms using Balsa-II. *Computer*, 21(5):14–36.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., and Stoodley, I. (2004). Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University. *Journal of Information Technology Education*, 3:143–160.

Bruce, C. and McMahon, C. (2002). Contemporary Developments in Teaching and Learning Introductory Programming: Towards a Research Proposal. Teaching and Learning Report 2002–2, Faculty of Information Technology, Queensland University of Technology.
URL `http://eprints.qut.edu.au/3232/`

Bruce, K. B. (2004). Controversy on How to Teach CS 1: A Discussion on the SIGCSE-Members Mailing List. *SIGCSE Bulletin*, 36(4):29–34.

Bruce-Lockhart, M. P., Crescenzi, P., and Norvell, T. S. (2009). Integrating Test Generation Functionality into The Teaching Machine Environment. *Electronic Notes in Theoretical Computer Science*, 224:115–124.

Bruce-Lockhart, M. P. and Norvell, T. S. (2000). Lifting the Hood of the Computer: Program Animation with The Teaching Machine. In: *Canadian Conference on Electrical and Computer Engineering*, volume 2 of *CCECE '00*, pp. 831–835. IEEE.

Bruce-Lockhart, M. P. and Norvell, T. S. (2007). Developing Mental Models of Computer Programming Interactively via the Web. In: *Proceedings of the 37th Annual Frontiers in Education Conference*, FIE '07, pp. S3H–3–S3H–8. IEEE.

Bruce-Lockhart, M. P., Norvell, T. S., and Cotronis, Y. (2007). Program and Algorithm Visualization in Engineering and Physics. *Electronic Notes in Theoretical Computer Science*, 178:111–119.

Bruner, J. S. (1960). *The Process of Education*. Harvard University Press.

Bruner, J. S. (1979). *On Knowing: Essays for the Left Hand*. Belknap Press.

Brusilovsky, P. and Loboda, T. D. (2006). WADEIn II: A Case for Adaptive Explanatory Visualization. *SIGCSE Bulletin*, 38(3):48–52.

Brusilovsky, P. L. (1992). Intelligent Tutor, Environment and Manual for Introductory Programming. *Educational and Training Technology International*, 29(1):26–34.

Buck, D. and Stucki, D. J. (2000). Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development. *SIGCSE Bulletin*, 32(1):75–79.

Burgess, G. A. (2005). Introduction to Programming: Blooming in America. *Journal of Computing in Small Colleges*, 21(1):19–28.

Burkhardt, J.-M., Détienne, F., and Wiedenbeck, S. (1997). Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension. In: S. Howard, J. Hammond, and G. Lindgaard (eds.), *Proceedings of the IFIP TC13 Interantional Conference on Human–Computer Interaction*, INTERACT '97, pp. 339–346. Chapman & Hall.

Burkhardt, J.-M., Détienne, F., and Wiedenbeck, S. (2002). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, 7(2):115–156.

Byckling, P. and Sajaniemi, J. (2005). Using Roles of Variables in Teaching: Effects on Program Construction. In: P. Romero, J. Good, E. Acosta-Chaparro, and S. Bryant (eds.), *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group*, PPIG'05, pp. 278–292. PPIG.

Byckling, P. and Sajaniemi, J. (2006a). A Role-Based Analysis Model for the Evaluation of Novices' Programming Knowledge Development. In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 85–96. ACM.

Byckling, P. and Sajaniemi, J. (2006b). Roles of Variables and Programming Skills Improvement. *SIGCSE Bulletin*, 38(1):413–417.

Cañas, J. J., Bajo, M. T., and Gonzalvo, P. (1994). Mental Models and Computer Programming. *International Journal of Human–Computer Studies*, 40(5):795–811.

Carbone, A., Hurst, J., Mitchell, I., and Gunstone, D. (2000). Principles for Designing Programming Exercises to Minimise Poor Learning Behaviours in Students. In: *Proceedings of the Australasian Conference on Computing Education*, ACE '00, pp. 26–33. ACM.

Carbone, A. and Sheard, J. (2003). Developing a Model of First Year Student Satisfaction in a Studio-Based Teaching Environment. *Journal of Information Technology Education*, 2:15–28.

Carlisle, M. C. (2009). Raptor: A Visual Programming Environment for Teaching Object-Oriented Programming. *Journal of Computing Sciences in Colleges*, 24(4):275–281.

Carter, J., Ala-Mutka, K., Fuller, U., Dick, M., English, J., Fone, W., and Sheard, J. (2003). How Shall We Assess This? *SIGCSE Bulletin*, 35(4):107–123.

Carter, J. and Jenkins, T. (1999). Gender and Programming: What's Going On? *SIGCSE Bulletin*, 31(3):1–4.

Cash, J. (1969). A Boy Named Sue (song). In: *At San Quentin*. Columbia Records.

Caspersen, M. E. (2007). *Educating Novices in the Skills of Programming*. Ph.D. thesis, Department of Computer Science, University of Aarhus.

Caspersen, M. E. and Bennedsen, J. (2007). Instructional Design of a Programming Course: A Learning Theoretic Approach. In: *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pp. 111–122. ACM.

Chang, K.-E., Chiao, B.-C., Chen, S.-W., and Hsiao, R.-S. (2000). A Programming Learning System for Beginners – A Completion Strategy Approach. *IEEE Transactions on Education*, 43(2):211–220.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, P., and Glaser, R. (1989). Learning from Examples via Self-Explanations. In: *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pp. 251–282. Routledge.

Chi, M. T. H., Glaser, R., and Rees, E. (1982). Expertise in Problem Solving. In: R. Sternberg (ed.), *Advances in the Psychology of Human Intelligence*, pp. 7–75. Lawrence Erlbaum.

Choi, J.-M. and Sato, K. (2006). Framing a Learning-Based Approach to Interactive System Design. In: K. Friedman, T. Love, E. Côrte-Real, and C. Rust (eds.), *Proceedings of the Design Research Society International Conference: Wonderground*. Centro Editorial do IADE.

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program. In: S. Fincher and M. Petre (eds.), *Computer Science Education Research*, pp. 85–100. Routledge.

Clancy, M., Stasko, J., Guzdial, M., Fincher, S., and Dale, N. (2001). Models and Areas for CS Education Research. *Computer Science Education*, 11(4):323–341.

Clark, R. E. (1982). Antagonism between Achievement and Enjoyment in ATI Studies. *Educational Psychologist*, 17(2):92–101.

Clear, T., Philpott, A., Robbins, P., and Simon (2009). Report on the Eighth BRACElet Workshop: BRACElet Technical Report 01/08. *Bulletin of Applied Computing and Information Technology*, 7(1).

Clear, T., Whalley, J., Robbins, P., Philpott, A., Eckerdal, A., Laakso, M.-J., and Lister, R. (2011). Report on the Final BRACElet Workshop: Auckland University of Technology, September 2010. *Journal of Applied Computing and Information Technology*, 15(1).

Coffield, F., Moseley, D., Hall, E., and Ecclestone, K. (2004). *Learning Styles and Pedagogy in Post-16 Learning: A Systematic and Critical Review*. Learning and Skills Research Centre.

Colburn, T. and Shute, G. (2007). Abstraction in Computer Science. *Minds and Machines*, 17(2):169–184.

Collins, A. B., Brown, J. S., and Newman, S. E. (1989). Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. In: L. B. Resnick (ed.), *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*. Lawrence Erlbaum.

Cooper, G., Tindall-Ford, S., Chandler, P., and Sweller, J. (2001). Learning by Imagining. *Journal of Experimental Psychology: Applied*, 7(1):68–82.

Cooper, S., Dann, W., and Pausch, R. (2003). Teaching Objects-First in Introductory Computer Science. *SIGCSE Bulletin*, 35(1):191–195.

Corney, M., Lister, R., and Teague, D. (2011). Early Relational Reasoning and the Novice Programmer: Swapping as the "*Hello World*" of Relational Reasoning. In: J. Hamer and M. de Raadt (eds.), *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, volume 114 of *CRPIT*, pp. 95–104. Australian Computer Society.

Corritore, C. L. and Wiedenbeck, S. (1991). What do Novices Learn During Program Comprehension? *International Journal of Human–Computer Interaction*, 3(2):199–222.

Corritore, C. L. and Wiedenbeck, S. (2001). An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers. *International Journal of Human–Computer Studies*, 54(1):1–23.

Cousin, G. (2006). An Introduction to Threshold Concepts. *Planet*, 17:4–5.

Craik, K. J. W. (1943). *The Nature of Explanation*. Cambridge University Press.

Cross, II, J. H., Barowski, L. A., Hendrix, D., Umphress, D., and Jain, J. (n.d.). jGRASP – An Integrated Development Environment with Visualizations for Improving Software Comprehensibility (web site). Accessed October 2011.
URL `http://www.jgrasp.org/`

Cross, II, J. H., Barowski, L. A., Hendrix, T. D., and Teate, J. C. (1996). Control Structure Diagrams for Ada 95. In: S. Carlson (ed.), *Proceedings of TRI-Ada'96: Disciplined Software Development with Ada*, pp. 143–147. ACM.

Cross, II, J. H., Hendrix, T. D., and Barowski, L. A. (2002). Using the Debugger as an Integral Part of Teaching CS1. In: *Proceedings of the 32nd Annual Frontiers in Education Conference*, FIE '02, pp. F1G–1–F1G–6. IEEE.

Cross, II, J. H., Hendrix, T. D., and Barowski, L. A. (2011). Combining Dynamic Program Viewing and Testing in Early Computing Courses. In: *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference*, COMPSAC '11, pp. 184–192. IEEE.

Cypher, A. (1993). *Watch What I Do: Programming by Demonstration*. MIT Press.

Davies, P. and Mangan, J. (2008). Embedding Threshold Concepts: From Theory to Pedagogical Principles to Learning Activities. In: R. Land and J. H. F. Meyer (eds.), *Threshold Concepts within the Disciplines*, pp. 37–50. SensePublishers.

de Kleer, J. and Brown, J. S. (1981). Mental Models of Physical Mechanisms and Their Acquisition. In: J. R. Anderson (ed.), *Cognitive Skills and Their Acquisition*, pp. 285–309. Lawrence Erlbaum.

de Kleer, J. and Brown, J. S. (1983). Assumptions and Ambiguities in Mechanistic Mental Models. In: D. Gentner and A. L. Stevens (eds.), *Mental Models*, pp. 155–190. Lawrence Erlbaum.

de Lipman, M. (1897). Nero, and the Burning of Rome (painting). In: *Quo Vadis*, by Henryk Sienkiewicz. Henry Altemus Company.

de Raadt, M. (2008). *Teaching Programming Strategies Explicitly to Novice Programmers*. Doctoral dissertation, University of Southern Queensland.

Decker, A. (2003). A Tale of Two Paradigms. *Journal of Computing in Small Colleges*, 19(2):238–246.

Decker, R. and Hirshfield, S. (1993). Top-Down Teaching: Object-Oriented Programming in CS 1. *SIGCSE Bulletin*, 25:270–273.

Dehnadi, S. and Bornat, R. (2006). The Camel has Two Humps. In: *Little PPIG*.
URL `http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf`

Deng, J. (2003). *Programming by Demonstration Environment for 1st Year Students*. Master's thesis, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989). Computing as a Discipline. *Communications of the ACM*, 32(1):9–23.

Denny, P., Luxton-Reilly, A., and Simon, B. (2008). Evaluating a New Exam Question: Parsons Problems. In: *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pp. 113–124. ACM.

Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. (2011). Understanding the Syntax Barrier for Novices. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pp. 208–212. ACM.

Derry, S. J. (1996). Cognitive Schema Theory in the Constructivist Debate. *Educational Psychologist*, 31(3&4):163–174.

Diamond, J. (1987). Soft Sciences are Often Harder than Hard Sciences. *Discover*, 8:34–39.
URL `http://bama.ua.edu/~sprentic/607%20Diamond%201987.htm`

Dijkstra, E. W., vs. al. (1989). A Debate on Teaching Computing Science [in response to Dijkstra's On the Cruelty of Really Teaching Computing Science]. *Communications of the ACM*, 32(12):1397–1414.

diSessa, A. A. (1993). Toward an Epistemology of Physics. *Cognition and Instruction*, 10(2):105–225.

diSessa, A. A. (2006). A History of Conceptual Change Research: Threads and Fault Lines. In: R. Sawyer (ed.), *The Cambridge Handbook of the Learning Sciences*, pp. 265–282. Cambridge University Press.

Dönmez, O. and İnceoğlu, M. M. (2008). A Web Based Tool for Novice Programmers: Interaction in Use. In: B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. O. Apduhan (eds.), *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA '08), Part I*, volume 6782 of *Lecture Notes in Computer Science*, pp. 530–540. Springer.

Dodani, M. H. (2003). Hello World! Goodbye Skills! *Journal of Object Technology*, 2(1):23–28.

Doukakis, D., Grigoriadou, M., and Tsaganou, G. (2007). Understanding the Programming Variable Concept with Animated Interactive Analogies. In: *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & its Applications Conference*, HERCMA '07.

Dreyfus, H. L. (1992). *What Computers Still Can't Do*. MIT Press.

Dreyfus, H. L. and Dreyfus, S. E. (2000). *Mind Over Machine*. Free Press.

Détienne, F. (1990). Expert Programming Knowledge: A Schema-based Approach. In: J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore (eds.), *Psychology of Programming*, pp. 205–222. Academic Press.

Détienne, F. (1997). Assessing the Cognitive Consequences of the Object-Oriented Approach: A Survey of Empirical Research on Object-Oriented Design by Individuals and Teams. *Interacting with Computers*, 9(1):47–72.

Détienne, F. and Soloway, E. (1990). An Empirically-Derived Control Structure for the Process of Program Understanding. *International Journal of Man–Machine Studies*, 33(3):323–342.

du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73.

du Boulay, B., O'Shea, T., and Monk, J. (1981). The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man–Machine Studies*, 14:237–249.

Ebel, G. and Ben-Ari, M. (2006). Affective Effects of Program Visualization. In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 1–5. ACM.

Eckerdal, A. (2006). *Novice Students' Learning of Object-Oriented Programming*. Licentiate's thesis, Department of Information Technology, Uppsala University.

Eckerdal, A., Laakso, M.-J., Lopez, M., and Sarkar, A. (2011). Relationship between Text and Action Conceptions of Programming: A Phenomenographic and Quantitative Perspective. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pp. 33–37. ACM.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2006a). Putting Threshold Concepts into Context in Computer Science Education. *SIGCSE Bulletin*, 38(3):103–107.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Zander, C. (2006b). Can Graduating Students Design Software Systems? *SIGCSE Bulletin*, 38(1):403–407.

Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., Thomas, L., and Zander, C. (2007). From Limen to Lumen: Computing Students in Liminal Spaces. In: *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pp. 123–132. ACM.

Eckerdal, A. and Thuné, M. (2005). Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. *SIGCSE Bulletin*, 37(3):89–93.

Eckerdal, A., Thuné, M., and Berglund, A. (2005). What Does it Take to Learn 'Programming Thinking'? In: *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pp. 135–142. ACM.

Ehlert, A. and Schulte, C. (2009). Empirical Comparison of Objects-First and Objects-Later. In: *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pp. 15–26. ACM.

Ehlert, A. and Schulte, C. (2010). Comparison of OOP First and OOP Later: First Results Regarding the Role of Comfort Level. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pp. 108–112. ACM.

Elliott Tew, A. (2010). *Assessing Fundamental Introductory Computing Concept Knowledge in a Language Independent Manner*. Ph.D. thesis, School of Interactive Computing, Georgia Institute of Technology.

Ellis, A., Carswell, L., Bernat, A., Deveaux, D., Frison, P., Meisalo, V., Meyer, J., Nulden, U., Rugelj, J., and Tarhio, J. (1998). Resources, Tools, and Techniques for Problem Based Learning in Computing. *SIGCUE Outlook*, 26(4):41–56.

Ericsson, K. A. and Kintsch, W. (1995). Long-Term Working Memory. *Psychological Review*, 102:211–245.

Ernest, P. (1995). The One and the Many. In: L. P. Steffe and J. E. Gale (eds.), *Constructivism in Education*, pp. 459–486. Lawrence Erlbaum.

Eskola, J. and Tarhio, J. (2002). On Visualization of Recursion with Excel. In: M. Ben-Ari (ed.), *Proceedings of the Second Program Visualization Workshop*, pp. 45–51. Department of Computer Science, University of Aarhus.

Esteves, M. and Mendes, A. J. (2003). OOP-Anim, A System to Support Learning of Basic Object Oriented Programming Concepts. In: *Proceedings of the 4th International Conference on Computer Systems and Technologies: e-Learning*, CompSysTech '03, pp. 573–579. ACM.

Esteves, M. and Mendes, A. J. (2004). A Simulation Tool to Help Learning of Object Oriented Programming Basics. In: *34th Annual Frontiers in Education Conference*, FIE '04. IEEE.

Etheredge, J. (2004). CMeRun: Program Logic Debugging Courseware for CS1/CS2 Students. *SIGCSE Bulletin*, 36(1):22–25.

Fekete, A. and Greening, A. (1996). Designing Closed Laboratories for a Computer Science Course. *SIGCSE Bulletin*, 28(1):295–299.

Felleisen, M., Bloch, S., Clements, J., Findler, R., Fisler, K., Flatt, M., Proulx, V., and Krishnamurthi, S. (n.d.). Program by Design (web site). Accessed October 2011.
URL `http://www.programbydesign.org/`

Fernández, A., Rossi, G., Morelli, P., Garcia Mari, L., Miranda, S., and Suarez, V. (1998). A Learning Environment to Improve Object-Oriented Thinking. In: *OOPSLA'98 Conference Proceedings*. ACM.

Fincher, S., et al. (n.d.). Share Project: Sharing & Representing Teaching Practice (web site). Accessed October 2011.
URL `http://www.sharingpractice.ac.uk/homepage.html`

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. (2002). DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, 12(2):159–182.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: Finding, Fixing and Flailing, a Multi-Institutional Study of Novice Debuggers. *Computer Science Education*, 18(2):93–116.

Flanagan, M. T. and Smith, J. (2008). From Playing to Understanding. In: R. Land and J. H. F. Meyer (eds.), *Threshold Concepts within the Disciplines*, pp. 91–103. SensePublishers.

Fleury, A. E. (1991). Parameter Passing: The Rules the Students Construct. *SIGCSE Bulletin*, 23(1):283–286.

Fleury, A. E. (2000). Programming in Java: Student-Constructed Rules. *SIGCSE Bulletin*, 32(1):197–201.

Ford, L. (1999). How Programmers Visualize Programs. Report R 271, Department of Computer Science, University of Exeter.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., and Thompson, E. (2007). Developing a Computer Science-Specific Learning Taxonomy. *SIGCSE Bulletin*, 39(4):152–170.

Gajraj, R. R., Williams, M., Bernard, M., and Singh, L. (2011). Transforming Source Code Examples into Programming Tutorials. In: *The Sixth International Multi-Conference on Computing in the Global Information Technology*, ICCGI 2011, pp. 160–164. IARIA.

Gallego-Carrillo, M., Gortázar-Bellas, F., and Velázquez-Iturbide, J. Á. (2004). JavaMod: An Integrated Java Model for Java Software Visualization. In: A. Korhonen (ed.), *Proceedings of the Third Program Visualization Workshop*, pp. 102–109. University of Warwick.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garner, S. (2002). COLORS for Programming: A System to Support the Learning of Programming. In: *Proceedings of the 2002 Informing Science and Information Technology Conference*, InSITE '02, pp. 533–542. Informing Science Institute.

Garner, S., Haden, P., and Robins, A. (2005). My Program is Correct but it Doesn't Run: a Preliminary Investigation of Novice Programmers' Problems. In: *Proceedings of the 7th Australasian Conference on Computing Education*, ACE '05, pp. 173–180. Australian Computer Society.

Gentner, D. and Gentner, D. R. (1983). Flowing Waters or Teeming Crowds: Mental Models of Electricity. In: D. Gentner and A. L. Stevens (eds.), *Mental Models*, pp. 99–130. Lawrence Erlbaum.

Gentner, D. and Stevens, A. L. (1983). *Mental Models*. Lawrence Erlbaum.

George, C. E. (2000a). EROSI – Visualising Recursion and Discovering New Errors. *SIGCSE Bulletin*, 32(1):305–309.

George, C. E. (2000b). Evaluating a Pedagogic Innovation: Execution Models & Program Construction Ability. In: *Proceedings of the 1st Annual Conference of the LTSN Centre for Information and Computer Sciences*, pp. 98–103.

George, C. E. (2000c). Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models. In: A. F. Blackwell and E. Bilotta (eds.), *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*, PPIG'00, pp. 33–44. PPIG.

George, C. E. (2002). Using Visualization to Aid Program Construction Tasks. *SIGCSE Bulletin*, 34(1):191–195.

Gergen, K. J. (1995). Social Construction and the Educational Process. In: L. P. Steffe and J. E. Gale (eds.), *Constructivism in Education*, pp. 17–39. Lawrence Erlbaum.

Gerjets, P. and Scheiter, K. (2003). Goal Configurations and Processing Strategies as Moderators Between Instructional Design and Cognitive Load: Evidence from Hypertext-Based Instruction. *Educational Psychologist*, 38(1):33–41.

Gestwicki, P. and Jayaraman, B. (2005). Methodology and Architecture of JIVE. In: *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis'05, pp. 95–104. ACM.

Gilligan, D. (1998). *An Exploration of Programming by Demonstration in the Domain of Novice Programming*. Master's thesis, School of Mathematics, Statistics and Computer Science, Victoria University of Wellington.

Glaser, R. and Chi, M. T. H. (1988). Overview. In: M. T. H. Chi, R. Glaser, and M. J. Farr (eds.), *The Nature of Expertise*, pp. xv–xxviii. Lawrence Erlbaum.

Gloor, P. A. (1998). User Interface Issues for Algorithm Animation. In: J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price (eds.), *Software Visualization: Programming as a Multimedia Experience*, chapter 11, pp. 145–152. MIT Press.

Gluga, R., Kay, J., Lister, R., Kleitman, S., and Lever, T. (2011). Over Confidence and Confusion in using Bloom for Programming Fundamentals Assessment. Technical Report 681, School of Information Technologies, The University of Sydney.
URL `http://sydney.edu.au/engineering/it/research/tr/tr681.pdf`

Gómez-Martín, P. P., Gómez-Martín, M. A., Díaz-Agudo, B., and González-Calero, P. A. (2005). Opportunities for CBR in Learning by Doing. In: H. Muñoz-Avila and F. Ricci (eds.), *Proceedings of the 6th International Conference on Case-Based Reasoning (ICCBR '05)*, volume 3620 of *Lecture Notes in Computer Science*, pp. 267–281. Springer.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., and Zilles, C. (2008). Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. *SIGCSE Bulletin*, 40(1):256–260.

Gómez-Martín, M. A., Gómez-Martín, P. P., and González-Calero, P. A. (2006). Dynamic Binding is the Name of the Game. In: R. Harper, M. Rauterberg, and M. Combetto (eds.), *Entertainment Computing – ICEC 2006*, volume 4161 of *Lecture Notes in Computer Science*, pp. 229–232. Springer.

Gondi, R. (2011). *Role of the Best Practices from Extant Literature in Current Algorithm and Data Structure Visualizations*. Master's thesis, Auburn University.

Gondow, K., Fukuyasu, N., and Arahori, Y. (2010). MieruCompiler: Integrated Visualization Tool with "Horizontal Slicing" for Educational Compilers. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 7–11. ACM.

Gonzalez, G. (2004). Constructivism in an Introduction to Programming Course. *Journal of Computing in Small Colleges*, 19(4):299–305.

Gorard, S. (2010). Research Design, as Independent of Methods. In: A. Tashakkori and C. Teddlie (eds.), *Sage Handbook of Mixed Methods in Social & Behavioral Research*, pp. 237–251. Sage, 2nd edition.

Gračanin, D., Matković, K., and Eltoweissy, M. (2005). Software Visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230.

Gray, J., Boyle, T., and Smith, C. (1998). A Constructivist Learning Environment Implemented in Java. *SIGCSE Bulletin*, 30(3):94–97.

Green, T. R. G. (2000). Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '00, pp. 21–28. ACM.

Green, T. R. G. and Blackwell, A. F. (1998). Cognitive Dimensions of Information Artefacts: A Tutorial. Accessed October 2011.
URL `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`

Green, T. R. G., Blandford, A. E., Church, L., Roast, C. R., and Clarke, S. (2006). Cognitive Dimensions: Achievements, New Directions, and Open Questions. *Journal of Visual Languages & Computing*, 17(4):328–365.

Green, T. R. G. and Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174.

Greening, T. (1998). Scaffolding for Success in Problem-Based Learning. *Medical Education Online*, 3:3–4.

Greening, T. (1999). Emerging Constructivist Forces in Computer Science Education: Shaping a New Future. In: T. Greening (ed.), *Computer Science Education in the 21st Century*, pp. 47–80. Springer.

Greening, T. and Kay, J. (2001). Editorial. *Computer Science Education*, Special Issue on Constructivism, 11(3):189–202.

Greeno, J. G. (1994). Gibson's Affordances. *Psychological Review*, 101(2):336–342.

Gries, D. (2008). A Principled Approach to Teaching OO First. *SIGCSE Bulletin*, 40(1):31–35.

Gries, P. and Gries, D. (2002). Frames and Folders: A Teachable Memory Model for Java. *Journal of Computing Sciences in Colleges*, 17(6):182–196.

Gries, P., Mnih, V., Taylor, J., Wilson, G., and Zamparo, L. (2005). Memview: A Pedagogically-Motivated Visual Debugger. In: *Proceedings of the 35th Annual Frontiers in Education Conference*, FIE '05, pp. 11–16. IEEE.

Götschi, T., Sanders, I., and Galpin, V. (2003). Mental Models of Recursion. *SIGCSE Bulletin*, 35(1):346–350.

Guindon, R., Krasner, H., and Curtis, B. (1987). Breakdowns and Processes during the Early Activities of Software Design by Professionals. In: G. M. Olson, S. Sheppard, and E. Soloway (eds.), *Empirical Studies of Programmers: Second Workshop*, pp. 65–82. Ablex Publishing.

Gunstone, R. F. (2000). Constructivism and Learning Research in Science Education. In: D. C. Phillips (ed.), *Constructivism in Education*, pp. 254–280. The National Society For The Study Of Education.

Guo, P. (n.d.). Online Python Tutor – Learn and Practice Python Programming in Your Web Browser. Accessed October 2011.
URL `http://people.csail.mit.edu/pgbovine/python/`

Guzdial, M. (2008). Paving the Way for Computational Thinking. *Communications of the ACM*, 51(8):25–27.

Guzdial, M. (2011). From Science to Engineering – Exploring the Dual Nature of Computing Education Research. *Communications of the ACM*, 54(2):37–39.

Guzdial, M. and Elliott Tew, A. (2006). Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education. In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 51–58. ACM.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P. (1997). Animation of User Algorithms on the Web. In: *Proceedings of Symposium on Visual Languages*, pp. 360–367. IEEE.

Hadjerrouit, S. (1998). A Constructivist Framework for Integrating the Java Paradigm into the Undergraduate Curriculum. *SIGCSE Bulletin*, 30(3):105–107.

Hamer, J., Cutts, Q., Jackova, J., Luxton-Reilly, A., McCartney, R., Purchase, H., Riedesel, C., Saeli, M., Sanders, K., and Sheard, J. (2008). Contributing Student Pedagogy. *SIGCSE Bulletin*, 40(4):194–212.

Hammer, D. (1996). Misconceptions or P-Prims: How May Alternative Perspectives of Cognitive Structure Influence Instructional Perceptions and Intentions? *Journal of the Learning Sciences*, 5(2):97–127.

Harlow, S., Cummings, R., and Aberasturi, S. M. (2006). Karl Popper and Jean Piaget: A Rationale for Constructivism. *The Educational Forum*, 71(1):41–48.

Hasselgren, B., Nordieng, T., and Österlund, A. (n.d.). Land of Phenomenography. Accessed March 2011. URL `http://www.ped.gu.se/biorn/phgraph/welcome.html`

Hattie, J. and Purdie, N. (1998). The SOLO Model: Addressing Fundamental Measurement Issues. In: B. Dart and G. Boulton-Lewis (eds.), *Teaching and Learning in Higher Education*, pp. 145–176. Australian Council for Educational Research.

Hauswirth, M., Jazayeri, M., and Winzer, A. (1998). A Java-Based Environment for Teaching Programming Language Concepts. In: *Proceedings of the 28th Annual Frontiers in Education Conference*, FIE '98, pp. 296–300. IEEE.

Helminen, J. (2009). *Jype – An Education-Oriented Integrated Program Visualization, Visual Debugging, and Programming Exercise Tool for Python*. Master's thesis, Department of Computer Science and Engineering, Helsinki University of Technology.

Henriksen, P. (2007). SIGCSE 2007 DC Application. Accessed October 2011. URL `http://www.cs.kent.ac.uk/archive/people/staff/ph53/SIGCSE2007DCApplication--PoulHenriksen.html`

Hintzman, D. L. (1986). "Schema Abstraction" in a Multiple-Trace Memory Model. *Psychological Review*, 93(4):411–428.

Hmelo, C. E. and Guzdial, M. (1996). Of Black and Glass Boxes: Scaffolding for Doing and Learning. In: *Proceedings of the 1996 International Conference on Learning Sciences*, ICLS '96, pp. 128–134. International Society of the Learning Sciences.

Hmelo-Silver, C. E., Duncan, R. G., and Chinn, C. A. (2007). Scaffolding and Achievement in Problem-Based and Inquiry learning: A Response to Kirschner, Sweller, and Clark (2006). *Educational Psychologist*, 42(2):99–107.

Holland, S., Griffiths, R., and Woodman, M. (1997). Avoiding Object Misconceptions. *SIGCSE Bulletin*, 29(1):131–134.

Holliday, M. A. and Luginbuhl, D. (2003). Using Memory Diagrams When Teaching a Java-Based CS1. In: *Proceedings of the 41st Annual ACM Southeast Conference*, pp. 376–381. ACM.

Holliday, M. A. and Luginbuhl, D. (2004). CS1 Assessment Using Memory Diagrams. *SIGCSE Bulletin*, 36(1):200–204.

Holloway, M., Alpay, E., and Bull, A. (2010). A Quantitative Approach to Identifying Threshold Concepts in Engineering Education. In: *Engineering Education 2010 Conference*, EE 2010. The Higher Education Academy Engineering Subject Centre.

Howe, E., Thornton, M., and Weide, B. W. (2004). Components-First Approaches to CS1/CS2: Principles and Practice. *SIGCSE Bulletin*, 36(1):291–295.

Howe, K. R. and Berv, J. (2000). Constructing Constructivism, Epistemological and Pedagogical. In: D. C. Phillips (ed.), *Constructivism in Education*, pp. 19–40. The National Society For The Study Of Education.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. *SIGCSE Bulletin*, 35(1):153–156.

Hu, M., Winikoff, M., and Cranefield, S. (2012). Teaching Novice Programming Using Goals and Plans in a Visual Notation. In: M. de Raadt and A. Carbone (eds.), *Proceedings of the 14th Australasian Conference on Computing Education (ACE '12)*, volume 123 of *CRPIT*, pp. 43–52. Australian Computer Society.

Hundhausen, C. D. (2002). Integrating Algorithm Visualization Technology into an Undergraduate Algorithms Course: Ethnographic Studies of a Social Constructivist Approach. *Computers & Education*, 39(3):237–260.

Hundhausen, C. D. (2005). Using End-User Visualization Environments to Mediate Conversations: A Communicative Dimensions' Framework. *Journal of Visual Languages & Computing*, 16(3):153–185.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of Recent Systems for Automatic Assessment of Programming Assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pp. 86–93. ACM.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. (1997). Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *SIGPLAN Notices*, 32(10):318–326.

Ireland, J., Tambyah, M. M., Neofa, Z., and Harding, T. (2009). The Tale of Four Researchers: Trials and Triumphs from the Phenomenographic Research Specialization. In: *AARE 2008 International Education Conference*. Australian Association for Research in Education.

Isoda, S., Shimomura, T., and Ono, Y. (1987). VIPS: A Visual Debugger. *IEEE Software*, 4(3):8–19.

Isohanni, E. and Knobelsdorf, M. (2010). Behind the Curtain: Students' Use of VIP After Class. In: *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pp. 87–96. ACM.

Isohanni, E. and Knobelsdorf, M. (2011). Students' Long-Term Engagement with the Visualization Tool VIP. In: A. Korhonen and R. McCartney (eds.), *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pp. 33–38. ACM.

Jeffries, R., Turner, A. A., Polson, P. G., and Atwood, M. E. (1981). The Processes Involved in Designing Software. In: J. R. Anderson (ed.), *Cognitive Skills and Their Acquisition*, pp. 255–283. Lawrence Erlbaum.

Jiménez-Díaz, G., Gómez-Albarrán, M., and González-Calero, P. A. (2008). Role-Play Virtual Environments: Recreational Learning of Software Design. In: *Proceedings of the 3rd European conference on Technology Enhanced Learning: Times of Convergence: Technologies Across Learning Contexts*, EC-TEL '08, pp. 27–32. Springer.

Jiménez-Díaz, G., González-Calero, P. A., and Gómez-Albarrán, M. (2011). Role-Play Virtual Worlds for Teaching Object-Oriented Design: The ViRPlay Development Experience. *Software – Practice and Experience*, 42(2):235–253.

Jiménez-Díaz, G., Gómez-Albarrán, M., Gómez-Martín, M. A., and González-Calero, P. A. (2005). Software Behaviour Understanding Supported by Dynamic Visualization and Role-Play. *SIGCSE Bulletin*, 37(3):54–58.

Jiménez-Peris, R., Patiño-Martínez, M., and Pacios-Martínez, J. (1999). VisMod: A Beginner-Friendly Programming Environment. In: *Proceedings of the 1999 ACM Symposium on Applied Computing*, SAC '99, pp. 115–120. ACM.

Johansson, B., Marton, F., and Svensson, L. (1985). An Approach to Describing Learning as Change between Qualitatively Different Conceptions. In: L. H. T. West and A. Pines (eds.), *Cognitive Structure and Conceptual Change*, pp. 233–257. Academic Press.

Johnson, C. G. and Fuller, U. (2006). Is Bloom's Taxonomy Appropriate for Computer Science? In: A. Berglund and M. Wiggberg (eds.), *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, Koli Calling 2006, pp. 120–123. Uppsala University.

Johnson, R. and Onwuegbuzie, A. J. (2004). Mixed Methods Research: A Research Paradigm Whose Time Has Come. *Educational Researcher*, 33(7):14–26.

Johnson-Laird, P. N. (1983). *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness*. Harvard University Press.

Jonassen, D. (2009). Reconciling a Human Cognitive Architecture. In: S. Tobias and T. M. Duffy (eds.), *Constructivist Instruction: Success or Failure?*, pp. 13–33. Taylor & Francis.

Jonassen, D. H. and Henning, P. (1996). Mental Models: Knowledge in the Head and Knowledge in the World. In: *Proceedings of the 1996 International Conference on Learning Sciences*, ICLS '96, pp. 433–438. International Society of the Learning Sciences.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. (2010). Identifying Student Misconceptions of Programming. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 107–111. ACM.

Kahney, H. (1983). What do Novice Programmers Know about Recursion? In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pp. 235–239. ACM.

Kaila, E. (2008). Ohjelmoinnin opetus ja opettajien suhtautuminen opetusta kehittäviin välineisiin. A survey of Finnish university teachers on the teaching of programming and tool adoption, in Finnish. Accessed October 2011.
URL `http://www.cs.hut.fi/Research/COMPSER/Verkostohanke/raportti.pdf`

Kaila, E., Laakso, M.-J., Rajala, T., and Salakoski, T. (2009a). Evaluation of Learner Engagement in Program Visualization. In: V. Uskov (ed.), *12th IASTED International Conference on Computers and Advanced Technology in Education*, CATE '09. ACTA Press.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. (2008). Automatic Assessment of Program Visualization Exercises. In: A. Pears and L. Malmi (eds.), *The 8th Koli Calling International Conference on Computing Education Research*, Koli Calling '08, pp. 105–108. Uppsala University.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. (2009b). Effects, Experiences and Feedback from Studies of a Program Visualization Tool. *Informatics in Education*, 8(1):17–34.

Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. (2010). Effects of Course-Long Use of a Program Visualization Tool. In: *Proceedings of the Twelfth Australasian Conference on Computing Education*, ACE '10, pp. 97–106. Australian Computer Society.

Kalyuga, S. (2005). Prior Knowledge Principle in Multimedia Learning. In: R. E. Mayer (ed.), *The Cambridge Handbook of Multimedia Learning*, pp. 325–338. Cambridge University Press.

Kalyuga, S. (2009). Knowledge Elaboration: A Cognitive Load Perspective. *Learning and Instruction*, 19:402–410.

Kalyuga, S. (2010). Schema Acquisition and Sources of Cognitive Load. In: J. L. Plass, R. Moreno, and R. Brünken (eds.), *Cognitive Load Theory*, pp. 48–64. Cambridge University Press.

Kalyuga, S. (2011). Cognitive Load Theory: How Many Types of Load Does It Really Need? *Educational Psychology Review*, 23:1–19.

Kalyuga, S., Ayres, P., Chandler, P., and Sweller, J. (2003). The Expertise Reversal Effect. *Educational Psychologist*, 38(1):23–31.

Kannusmäki, O., Moreno, A., Myller, N., and Sutinen, E. (2004). What a Novice Wants: Students Using Program Visualization in Distance Programming Course. In: A. Korhonen (ed.), *Proceedings of the Third Program Visualization Workshop*, pp. 126–133. University of Warwick.

Karavirta, V. (n.d.). XAAL: eXtensible Algorithm Animation Language. Accessed October 2011. URL http://xaal.org/

Kasmarik, K. and Thurbon, J. (2003). Experimental Evaluation of a Program Visualisation Tool for Use in Computer Science Education. In: *Proceedings of the Asia-Pacific Symposium on Information Visualisation – Volume 24*, APVis '03, pp. 111–116. Australian Computer Society.

Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J. H., and Crawford, K. (2000). Problem-Based Learning for Foundation Computer Science Courses. *Computer Science Education*, 10(2):109–128.

Kelleher, C. and Pausch, R. (2005). Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, 37(2):83–137.

Kempton, W. (1986). Two Theories of Home Heat Control. *Cognitive Science*, 10:75–90.

Kessel, C. J. and Wickens, C. D. (1982). The Transfer of Failure-Detection Skills between Monitoring and Controlling Dynamic Systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 24(1):49–60.

Kessler, C. M. and Anderson, J. R. (1986). Learning Flow of Control: Recursive and Iterative Procedures. *Human–Computer Interaction*, 2(2):135–166.

Khairuddin, N. N. and Hashim, K. (2008). Application of Bloom's Taxonomy in Software Engineering Assessments. In: *Proceedings of the 8th Conference on Applied Computer Science*, pp. 66–69. World Scientific and Engineering Academy and Society (WSEAS).

Khazaei, B. and Jackson, M. (2002). Is There Any Difference in Novice Comprehension of a Small Program Written in the Event-Driven and Object-Oriented Styles? In: *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pp. 19–26. IEEE.

Kinnunen, P. and Malmi, L. (2005). Problems in Problem-Based Learning – Experiences, Analysis and Lessons Learned on an Introductory Programming Course. *Informatics in Education*, 4(2):193.

Kirby, S., Toland, B., and Deegan, C. (2010). Program Visualisation Tool for Teaching Programming in C. In: *Proceedings of the International Conference on Education, Training and Informatics*, ICETI 2010. International Institute of Informatics and Systemics.

Kirschner, F. (2009). *United Brains for Complex Learning: A Cognitive-Load Approach to Collaborative Learning Efficiency*. Doctoral dissertation, Open Universiteit Nederland.

Kirschner, P. A., Sweller, J., and Clark, R. E. (2006). Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist*, 41(2):75–86.

Klein, G. A. (1999). *Sources of Power: How People Make Decisions*. MIT Press.

Knobelsdorf, M. (2008). A Typology of CS Students' Preconditions for Learning. In: A. Pears and L. Malmi (eds.), *The 8th Koli Calling International Conference on Computing Education Research*, pp. 62–71. Uppsala University.

Knobelsdorf, M. and Schulte, C. (2007). Computer Science in Context – Pathways to Computer Science. In: R. Lister and Simon (eds.), *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 65–76. Australian Computer Society.

Knuth, D. E. (1989). The Errors of TEX. *Software – Practice and Experience*, 19(7):607–685.

Ko, A. J. and Myers, B. A. (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84.

Ko, P. Y. and Marton, F. (2004). Variation and the Secret of the Virtuoso. In: F. Marton and A. B. M. Tsui (eds.), *Classroom Discourse and the Space of Learning*, pp. 43–62. Lawrence Erlbaum.

Kölling, M. (2008). Using BlueJ to Introduce Programming. In: J. Bennedsen, M. E. Caspersen, and M. Kölling (eds.), *Reflections on the Teaching of Programming: Methods and Implementations*, pp. 98–115. Springer.

Kölling, M. and Barnes, D. J. (2008). Apprentice-Based Learning Via Integrated Lectures and Assignments. In: J. Bennedsen, M. E. Caspersen, and M. Kölling (eds.), *Reflections on the Teaching of Programming: Methods and Implementations*, pp. 17–29. Springer.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ System and Its Pedagogy. *Journal of Computer Science Education*, 13(4).

Kollmansberger, S. (2010). Helping Students Build a Mental Model of Computation. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pp. 128–131. ACM.

Korhonen, A. (2003). *Visual Algorithm Simulation*. Doctoral dissertation, Department of Computer Science and Engineering, Helsinki University of Technology.

Korhonen, A. (n.d.). Ohjelmoinnin perusopetuksen verkosto. A web site for networking amongst Finnish teachers of introductory programming. In Finnish. Accessed October 2011.
URL `http://www.cs.hut.fi/Research/COMPSER/Verkostohanke/index.shtml`

Korhonen, A., Helminen, J., Karavirta, V., and Seppälä, O. (2009a). TRAKLA2. In: A. Pears and C. Schulte (eds.), *The 9th Koli Calling International Conference on Computing Education Research*, Koli Calling '09, pp. 43–46.

Korhonen, A., Laakso, M.-J., and Myller, N. (2009b). How Does Algorithm Visualization Affect Collaboration? Video Analysis of Engagement and Discussions. In: J. Filipe and J. Cordeiro (eds.), *Proceedings of the 5th International Conference on Web Information Systems and Technologies*, WEBIST '09, pp. 479–488. Institute for Systems and Technologies of Information, Control and Communication.

Korhonen, A., Malmi, L., Silvasti, P., Karavirta, V., Lönnberg, J., Nikander, J., Stålnacke, K., and Ihantola, P. (2004). Matrix – A Framework for Interactive Software Visualization. Research Report TKO-B 154/04, Department of Computer Science and Engineering, Helsinki University of Technology.

Korsh, J. F. and Sangwan, R. (1998). Animating Programs and Students in the Laboratory. In: *Proceedings of the 28th Annual Frontiers in Education Conference*, FIE '98, pp. 1139–1144. IEEE.

Kranch, D. A. (2011). Teaching the Novice Programmer: A Study of Instructional Sequences and Perception. *Education and Information Technologies*, Online First™.

Krathwohl, D. R. (2002). A Revision of Bloom's Taxonomy: An Overview. *Theory into Practice*, 41(4):212–218.

Krippendorff, K. (2003). *Content Analysis: An Introduction to Its Methodology*. Sage Publications, 2nd edition.

Kruger, J. and Dunning, D. (1999). Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134.

Kuhn, D. (2007). Is Direct Instruction an Answer to the Right Question? *Educational Psychologist*, 42(2):109–113.

Kuhn, T. S. (1962). *The Structure of Scientific Revolutions*. University of Chicago.

Kumar, A. N. (2005). Results from the Evaluation of the Effectiveness of an Online Tutor on Expression Evaluation. *SIGCSE Bulletin*, 37(1):216–220.

Kumar, A. N. (2009). Data Space Animation for Learning the Semantics of C++ Pointers. *SIGCSE Bulletin*, 41(1):499–503.

Kunkle, W. M. (2010). *The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts*. Ph.D. thesis, Drexel University.

Kurland, D., Pea, R. D., Clement, C., and Mawby, R. (1986). A Study of the Development of Programming Ability and Thinking Skills in High School Students. *Journal of Educational Computing Research*, 2(4):429–458.

Kvale, S. (1996). *InterViews: An Introduction to Qualitative Research Interviewing*. Sage Publications, 1st edition.

Laakso, M.-J., Myller, N., and Korhonen, A. (2009). Comparing Learning Performance of Students Using Algorithm Visualizations Collaboratively on Different Engagement Levels. *Journal of Educational Technology & Society*, 12(2):267–282.

Laakso, M.-J., Rajala, T., Kaila, E., and Salakoski, T. (2008). The Impact of Prior Experience in Using a Visualization Tool on Learning to Program. In: *Proceedings of Cognition and Exploratory Learning in Digital Age*, CELDA '08.

Lahtinen, E. and Ahoniemi, T. (2005). Visualizations to Support Programming on Different Levels of Cognitive Development. In: T. Salakoski, T. Mäntylä, and M. Laakso (eds.), *Proceedings of the Fifth Koli Calling Conference on Computer Science Education*, Koli Calling '05, pp. 87—94. Turku Centre for Computer Science.

Lahtinen, E. and Ahoniemi, T. (2007). Annotations for Defining Interactive Instructions to Interpreter Based Program Visualization Tools. *Electronic Notes in Theoretical Computer Science*, 178:121–128.

Lahtinen, E., Ahoniemi, T., and Salo, A. (2007a). Effectiveness of Integrating Program Visualizations to a Programming Course. In: R. Lister and Simon (eds.), *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 195–198. Australian Computer Society.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A Study of the Difficulties of Novice Programmers. *SIGCSE Bulletin*, 37(3):14–18.

Lahtinen, E., Järvinen, H.-M., and Melakoski-Vistbacka, S. (2007b). Targeting Program Visualizations. *SIGCSE Bulletin*, 39(3):256–260.

Land, R. and Meyer, J. H. F. (eds.) (2008). *Threshold Concepts within the Disciplines*. SensePublishers.

Larochelle, M. and Bednarz, N. (1998). Constructivism and Education: Beyond Epistemological Correctness. In: M. Larochelle, N. Bednarz, and J. Garrison (eds.), *Constructivism and Education*, pp. 1–20. Cambridge University Press.

Larochelle, M., Bednarz, N., and Garrison, J. (eds.) (1998). *Constructivism and Education*. Cambridge University Press.

Lattu, M., Meisalo, V., and Tarhio, J. (2003). A Visualisation Tool as a Demonstration Aid. *Computers & Education*, 41(2):133–148.

Lattu, M., Tarhio, J., and Meisalo, V. (2000). How a Visualization Tool Can Be Used – Evaluating a Tool in a Research & Development Project. In: A. F. Blackwell and E. Bilotta (eds.), *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group*, PPIG'00, pp. 19–32. PPIG.

Lauer, T. (2006). Learner Interaction with Algorithm Visualizations: Viewing vs. Changing vs. Constructing. *SIGCSE Bulletin*, 38(3):202–206.

Lave, J. and Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press.

Lego Group (n.d.). Lego Mindstorms (web site). Accessed October 2011.
URL `http://mindstorms.lego.com/`

Lessa, D., Czyz, J. K., Gestwicki, P. V., and Jayaraman, B. (n.d.). JIVE: Java Interactive Visualization Environment (web site). Accessed April 2012.
URL `http://www.cse.buffalo.edu/jive/`

Letovsky, S. and Soloway, E. (1986). Delocalized Plans and Program Comprehension. *Software*, 3(3):41–49.

Lewis, J. (2000). Myths about Object-Orientation and Its Pedagogy. *SIGCSE Bulletin*, 32(1):245–249.

Lincoln, Y. S. and Guba, E. G. (1985). *Naturalistic Inquiry*. Sage Publications.

Lincoln, Y. S., Lynham, S. A., and Guba, E. G. (2011). Paradigmatic Controversies, Contradictions, and Emerging Confluences, Revisited. In: N. K. Denzin and Y. S. Lincoln (eds.), *The Sage Handbook of Qualitative Research*, pp. 97–128. Sage, 4th edition.

Linn, M. C. and Clancy, M. J. (1992). The Case for Case Studies of Programming Problems. *Communications of the ACM*, 35(3):121–132.

Linn, M. C. and Dalbey, J. (1985). Cognitive Consequences of Programming Instruction: Instruction, Access, and Ability. *Educational Psychologist*, 20(4):191–206.

Lister, R. (2001). Objectives and Objective Assessment in CS1. *SIGCSE Bulletin*, 33(1):292–296.

Lister, R. (2004). Teaching Java First: Experiments with a Pigs-Early Pedagogy. In: *Proceedings of the Sixth Australasian Conference on Computing Education*, ACE '04, pp. 177–183. Australian Computer Society.

Lister, R. (2011a). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. In: J. Hamer and M. de Raadt (eds.), *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, volume 114 of *CRPIT*, pp. 9–18. Australian Computer Society.

Lister, R. (2011b). Programming, Syntax and Cognitive Load. *ACM Inroads*, 2(2):21–22.

Lister, R. (2011c). Programming, Syntax and Cognitive Load (Part 2). *ACM Inroads*, 2(3):16–17.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin*, 36(4):119–150.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., and Whalley, J. L. (2006a). Research Perspectives on the Objects-Early Debate. *SIGCSE Bulletin*, 38(4):146–165.

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., and Thompson, E. (2009a). Naturally Occurring Data as Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *SIGCSE Bulletin*, 41(4):156–173.

Lister, R., Fidge, C., and Teague, D. (2009b). Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming. *SIGCSE Bulletin*, 41(3):161–165.

Lister, R. and Leaney, J. (2003). Introductory Programming, Criterion-Referencing, and Bloom. *SIGCSE Bulletin*, 35(1):143–147.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006b). Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. *SIGCSE Bulletin*, 38(3):118–122.

Lönnberg, J. (2012). *Understanding and Debugging Concurrent Programs through Visualisation*. Doctoral dissertation, Department of Computer Science and Engineering, Aalto University.

Loftus, C., Thomas, L., and Zander, C. (2011). Can Graduating Students Design: Revisited. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pp. 105–110. ACM.

Lopez, M., Whalley, J., Robbins, P., and Lister, R. (2008). Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In: *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pp. 101–112. ACM.

Lucas, U. and Mladenovic, R. (2006). Developing New 'World Views': Threshold Concepts in Introductory Accounting. In: J. H. F. Meyer and R. Land (eds.), *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, pp. 148–159. Routledge.

Lui, A. K., Kwan, R., Poon, M., and Cheung, Y. H. Y. (2004). Saving Weak Programming Students: Applying Constructivism in a First Programming Course. *SIGCSE Bulletin*, 36(2):72–76.

Ma, L. (2007). *Investigating and Improving Novice Programmers' Mental Models of Programming Concepts*. Ph.D. thesis, Department of Computer & Information Sciences, University of Strathclyde.

Ma, L., Ferguson, J., Roper, M., and Wood, M. (2011). Investigating and Improving the Models of Programming Concepts Held by Novice Programmers. *Computer Science Education*, 21(1):57–80.

Ma, L., Ferguson, J. D., Roper, M., Ross, I., and Wood, M. (2009). Improving the Mental Models Held by Novice Programmers using Cognitive Conflict and Jeliot Visualisations. *SIGCSE Bulletin*, 41(3):166–170.

Madison, S. and Gifford, J. (1997). Parameter Passing: The Conceptions Novices Construct. Research report.
URL `http://eric.ed.gov/PDFS/ED406211.pdf`

Maletic, J. I., Marcus, A., and Collard, M. L. (2002). A Task Oriented View of Software Visualization. In: *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '02, pp. 32–40. IEEE.

Malmi, L., Sheard, J., Simon, Bednarik, R., Helminen, J., Korhonen, A., Myller, N., Sorva, J., and Taherkhani, A. (2010). Characterizing Research in Computing Education: A Preliminary Analysis of the Literature. In: *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pp. 3–12. ACM.

Mann, L. M., Linn, M. C., and Clancy, M. (1994). Can Tracing Tools Contribute to Programming Proficiency? The LISP Evaluation Modeler. *Interactive Learning Environments*, 4(1):96–113.

Maravić Čisar, S., Pinter, R., Radosav, D., and Čisar, P. (2010). Software Visualization: The Educational Tool to Enhance Student Learning. In: *Proceedings of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics*, MIPRO '10, pp. 990–994. IEEE.

Maravić Čisar, S., Radosav, D., Pinter, R., and Čisar, P. (2011). Effectiveness of Program Visualization in Learning Java: A Case Study with Jeliot 3. *International Journal of Computers Communications & Control*, 6(4):669–682.

Markman, A. B. (1999). *Knowledge Representation*. Lawrence Erlbaum.

Markman, A. B. and Gentner, D. (2001). Thinking. *Annual Review of Psychology*, 52:223–247.

Marton, F. (1986). Phenomenography – A Research Approach to Investigating Different Understandings of Reality. *Journal of Thought*, 21(3):28–49.

Marton, F. (1993). Our Experience of the Physical World. *Cognition and Instruction*, 10(2):227–237.

Marton, F. (2000). The Structure of Awareness. In: J. A. Bowden and E. Walsh (eds.), *Phenomenography*, pp. 102–116. RMIT University Press.

Marton, F. (2007). Towards a Pedagogical Theory of Learning. In: N. Entwistle and P. Tomlinson (eds.), *Student Learning and University Teaching*, pp. 19–30. British Psychological Society.

Marton, F. and Booth, S. (1997). *Learning and Awareness*. Lawrence Erlbaum.

Marton, F. and Pong, W. Y. (2005). On the Unit of Description in Phenomenography. *Higher Education Research & Development*, 24(4):335–348.

Marton, F., Runesson, U., and Tsui, A. B. M. (2004). The Space of Learning. In: F. Marton and A. B. M. Tsui (eds.), *Classroom Discourse and the Space of Learning*, pp. 3–40. Lawrence Erlbaum.

Marton, F. and Säljö, R. (1976). On Qualitative Differences in Learning: I – Outcome and Process. *British Journal of Educational Psychology*, 46(1):4–11.

Marton, F. and Tsui, A. B. M. (eds.) (2004). *Classroom Discourse and the Space of Learning*. Lawrence Erlbaum.

Mason, R. and Cooper, G. (2012). Why the Bottom 10% Just Can't Do It – Mental Effort Measures and Implication for Introductory Programming Courses. In: M. de Raadt and A. Carbone (eds.), *Proceedings of the 14th Australasian Conference on Computing Education (ACE '12)*, volume 123 of *CRPIT*, pp. 187–196. Australian Computer Society.

Matthews, M. R. (1992). Old Wine in New Bottles: A Problem with Constructivist Epistemology. In: H. Alexander (ed.), *Proceedings of the 48th Annual Philosophy of Education Meeting*, pp. 303–311. Philosophy of Education Society.

Matthews, M. R. (1994). *Science Teaching: The Role of History and Philosophy of Science*. Routledge.

Matthews, M. R. (2000). Appraising Constructivism in Science and Mathematics. In: D. C. Phillips (ed.), *Constructivism in Education*, pp. 161–192. The National Society For The Study Of Education.

Mayer, R. E. (1975). Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models. *Journal of Educational Psychology*, 67(6):725–734.

Mayer, R. E. (1979). A Psychology of Learning BASIC. *Communications of the ACM*, 22(11):589–593.

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1):121–141.

Mayer, R. E. (1985). Learning in Complex Domains: A Cognitive Analysis of Computer Programming. In: *The Psychology of Learning and Motivation: Advances in Research and Theory*, volume 19, pp. 89–130. Academic Press.

Mayer, R. E. (2004). Should There Be a Three-Strikes Rule Against Pure Discovery Learning? *American Psychologist*, 59(1):14–19.

Mayer, R. E. (ed.) (2005). *The Cambridge Handbook of Multimedia Learning*. Cambridge University Press.

Mayer, R. E. (2009). *Multimedia Learning*. Cambridge University Press, 2nd edition.

Mayer, R. E. and Alexander, P. A. (2011). *Handbook of Research on Learning and Instruction*. Routledge.

McCartney, R., Boustedt, J., Eckerdal, A., Moström, J. E., Sanders, K., Thomas, L., and Zander, C. (2009). Liminal Spaces and Learning Computing. *European Journal of Engineering Education*, 34(4):383–391.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: A Review of the Literature from an Educational Perspective. *Computer Science Education*, 18(2):67–92.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bulletin*, 33(4):125–180.

McCune, V. and Hounsell, D. (2005). The Development of Students' Ways of Thinking and Practising in Three Final-Year Biology Courses. *Higher Education*, 49(3):255–289.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., and Hirtle, S. C. (1981). Knowledge Organization and Skill Differences in Computer Programmers. *Cognitive Psychology*, 13:307–325.

Media Computation teachers (n.d.). Media Computation Teachers Website. Accessed October 2011. URL `http://coweb.cc.gatech.edu/mediaComp-teach`

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2010). Learning Computer Science Concepts with Scratch. In: *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pp. 69–76. ACM.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2011). Habits of Programming in Scratch. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pp. 168–172. ACM.

Menand, L. (1997). *Pragmatism: A Reader*. Vintage.

Meyer, J. H. F. and Land, R. (2003). Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and Practising within the Disciplines. In: C. Rust (ed.), *Improving Student Learning – Ten Years On*. Oxford Centre for Staff and Learning Development.

Meyer, J. H. F. and Land, R. (eds.) (2006). *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*. Routledge.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63:81–97.

Miller, L. A. (1981). Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Systems Journal*, 20(2):184–215.

Milne, I. and Rowe, G. (2002). Difficulties in Learning and Teaching Programming – Views of Students and Tutors. *Education and Information Technologies*, 7(1):55–66.

Milne, I. and Rowe, G. (2004). OGRE: Three-Dimensional Program Visualization for Novice Programmers. *Education and Information Technologies*, 9(3):219–237.

Minsky, M. (1974). A Framework for Representing Knowledge. AI Memo 306, Massachusetts Institute of Technology.

MIT Media Lab (n.d.). Scratch (web site). Accessed October 2011. URL `http://scratch.mit.edu/`

Miura, M., Sugihara, T., and Kunifuji, S. (2009). Anchor Garden: An Interactive Workbench For Basic Data Concept Learning in Object Oriented Programming Languages. *SIGCSE Bulletin*, 41(3):141–145.

Miyadera, Y., Kurasawa, K., Nakamura, S., Yonezawa, N., and Yokoyama, S. (2007). A Real-time Monitoring System for Programming Education using a Generator of Program Animation Systems. *Journal of Computers*, 2(3):12–20.

Moray, N. (1990). A Lattice Theory Approach to the Structure of Mental Models. *Philosophical Transactions of the Royal Society of London*, 327(1241):577–58.

Moreno, A. (2005). *The Design and Implementation of Intermediate Codes for Software Visualization*. Master's thesis, Department of Computer Science, University of Joensuu.

Moreno, A. and Joy, M. S. (2007). Jeliot 3 in a Demanding Educational Setting. *Electronic Notes in Theoretical Computer Science*, 178:51–59.

Moreno, A. and Myller, N. (2003). Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In: *Proceedings of the International Conference on Networked e-learning for European Universities*. EUROPACE.

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing Programs with Jeliot 3. In: M. F. Costabile (ed.), *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '04, pp. 373–376. ACM.

Moreno, R. (2006). When Worked Examples Don't Work: Is Cognitive Load Theory at an Impasse? *Learning and Instruction*, 16(2):170–181.

Morgan, D. L. (2007). Paradigms Lost and Pragmatism Regained. *Journal of Mixed Methods Research*, 1(1):48–76.

Moritz, S. H. and Blank, G. D. (2005). A Design-First Curriculum for Teaching Java in a CS1 Course. *SIGCSE Bulletin*, 37(2):89–93.

Moström, J. E., Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., and Zander, C. (2008). Concrete Examples of Abstraction as Manifested in Students' Transformative Experiences. In: *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pp. 125–136. ACM.

Moström, J. E., Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., and Zander, C. (2009). Computer Science Student Transformations: Changes and Causes. *SIGCSE Bulletin*, 41(3):181–185.

Mselle, L. J. (2011). Enhancing Comprehension by Using Random Access Memory (RAM) Diagrams in Teaching Programming: Class Experiment. In: *Proceedings of the 23rd Annual Workshop of the Psychology of Programming Interest Group*, PPIG'11. PPIG.

Mulholland, P. (1997). Using a Fine-Grained Comparative Evaluation Technique to Understand and Design Software Visualization Tools. In: *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ESP '97, pp. 91–108. ACM.

Muller, O. (2005). Pattern Oriented Instruction and the Enhancement of Analogical Reasoning. In: *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pp. 57–67. ACM.

Murphy, L., McCauley, R., and Fitzgerald, S. (2012). 'Explain in Plain English' Questions: Implications for Teaching. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pp. 385–390. ACM.

Myers, B. A. (1990). Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123.

Myers, B. A., Chandhok, R., and Sareen, A. (1988). Automatic Data Visualization for Novice Pascal Programmers. In: *Proceedings of the IEEE Workshop on Visual Languages*, pp. 192–198. IEEE.

Myller, N. and Bednarik, R. (2006). Methodologies for Studies of Program Visualization. In: *Methods, Materials and Tools for Programming Education*, MMT '06, pp. 37–42.

Myller, N., Bednarik, R., and Moreno, A. (2007a). Integrating Dynamic Program Visualization into BlueJ: The Jeliot 3 Extension. In: *Seventh IEEE International Conference on Advanced Learning Technologies*, ICALT '07, pp. 505–506. IEEE.

Myller, N., Bednarik, R., Sutinen, E., and Ben-Ari, M. (2009). Extending the Engagement Taxonomy: Software Visualization and Collaborative Learning. *ACM Transactions on Computing Education*, 9(1):1–27.

Myller, N., Laakso, M., and Korhonen, A. (2007b). Analyzing Engagement Taxonomy in Collaborative Algorithm Visualization. *SIGCSE Bulletin*, 39(3):251–255.

Najjar, L. J. (1998). Principles of Educational Multimedia User Interface Design. *Human Factors*, 40(2):311–323.

Naps, T. L. (2005). JHAVÉ: Supporting Algorithm Visualization. *Computer Graphics and Applications*, 25(5):49–55.

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. Á. (2003). Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bulletin*, 35(2):131–152.

Naps, T. L. and Stenglein, J. (1996). Tools for Visual Exploration of Scope and Parameter Passing in a Programming Languages Course. *SIGCSE Bulletin*, 28(1):305–309.

Nevalainen, S. and Sajaniemi, J. (2005). Short-Term Effects of Graphical versus Textual Visualisation of Variables on Program Perception. In: P. Romero, J. Good, E. Acosta-Chaparro, and S. Bryant (eds.), *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group*, PPIG'05, pp. 77–91. PPIG.

Nevalainen, S. and Sajaniemi, J. (2006). An Experiment on Short-Term Effects of Animated versus Static Visualization of Operations on Program Perception. In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 7–16. ACM.

Nevalainen, S. and Sajaniemi, J. (2008). An Experiment on the Short-Term Effects of Engagement and Representation in Program Animation. *Journal of Educational Computing Research*, 39(4):395–430.

Nicholson, A. E. and Fraser, K. M. (1996). Methodologies for Teaching New Programming Languages: a Case Study Teaching LISP. In: *Proceedings of the 2nd Australasian Conference on Computer Science Education*, ACSE '97, pp. 84–90. ACM.

Nielsen, J. (n.d.). Ten Usability Heuristics. Accessed October 2011.
URL `http://www.useit.com/papers/heuristic/heuristic_list.html`

Niiniluoto, I. (1980). *Johdatus tieteenfilosofiaan: käsitteen- ja teorianmuodostus*. Otava. "An Introduction to Philosophy of Science: Formation of Concepts and Theory", in Finnish.

Niiniluoto, I. (2002). *Critical Scientific Realism*. Oxford University Press.

Nikander, J., Helminen, J., and Korhonen, A. (2009). Experiences on Using TRAKLA2 to Teach Spatial Data Algorithms. *Electronic Notes in Theoretical Computer Science*, 224:77–88.

Nordström, M. and Börstler, J. (2011). Improving OO Example Programs. *IEEE Transactions on Education*.
URL `http://www8.cs.umu.se/~marie/thesis/files/PVII_HowTo_Examples.pdf`

Norman, D. A. (1983). Some Observations on Mental Models. In: D. Gentner and A. L. Stevens (eds.), *Mental Models*, pp. 7–14. Lawrence Erlbaum.

Norman, D. A. (2007). Simplicity is Highly Overrated. *Interactions*, 14(2):40–41.

Norman, D. A. (2008). Simplicity is Not the Answer. *Interactions*, 15(5):45–46.

Norman, G. R. and Schmidt, H. G. (1992). The Psychological Basis of Problem-Based Learning: A Review of the Evidence. *Academic Medicine*, 67(9):557–565.

Nuutila, E., Törmä, S., and Malmi, L. (2005). PBL and Computer Programming – The Seven Steps Method with Adaptations. *Computer Science Education*, 15(2):123–142.

Oechsle, R. and Morth, T. (2007). Peer Review of Animations Developed by Students. *Electronic Notes in Theoretical Computer Science*, 178:181–186.

Oechsle, R. and Schmitt, T. (2002). JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In: S. Diehl (ed.), *Revised Lectures on Software Visualization, International Seminar*, pp. 176–190. Springer.

O'Kelly, J. and Gibson, J. P. (2006). RoboCode & Problem-Based Learning: A Non-Prescriptive Approach to Teaching Programming. *SIGCSE Bulletin*, 38(3):217–221.

Oliver, D., Dobele, T., Greber, M., and Roberts, T. (2004). This Course has a Bloom Rating of 3.9. In: *Proceedings of the Sixth Australasian Conference on Computing Education*, ACE '04, pp. 227–231. Australian Computer Society.

Onwuegbuzie, A. J. and Johnson, R. B. (2006). The Validity Issue in Mixed Research. *Research in the Schools*, 13(1):48–63.

Özdemir, G. and Clark, D. B. (2007). An Overview of Conceptual Change Theories. *Eurasia Journal of Mathematics, Science & Technology Education*, 3(4):351–361.

Paas, F., Renkl, A., and Sweller, J. (eds.) (2003). *Educational Psychologist*. Special Issue: Cognitive Load Theory, 38(1).

Paas, F. G. W. C. and van Merriënboer, J. J. G. (1994). Variability of Worked Examples and Transfer of Geometrical Problem-Solving Skills: A Cognitive-Load Approach. *Journal of Educational Psychology*, 86(1):122–133.

Palumbo, D. B. (1990). Programming Language/Problem-Solving Research: A Review of Relevant Issues. *Review of Educational Research*, 60(1):65–89.

Pane, J. F., Ratanamahatana, C. A., and Myers, B. A. (2001). Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human–Computer Studies*, 54(2):237–264.

Pang, M. F. (2003). Two Faces of Variation: On Continuity in the Phenomenographic Movement. *Scandinavian Journal of Educational Research*, 47(2):145–156.

Papert, S. and Harel, I. (1991). *Constructionism*. Ablex Publishing.

Pareja-Flores, C., Urquiza-Fuentes, J., and Velázquez-Iturbide, J. Á. (2007). WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization. *SIGPLAN Notices*, 42(3):14–23.

Parker, J. R. and Becker, K. (2003). Measuring Effectiveness of Constructivist and Behaviourist Assignments in CS102. *SIGCSE Bulletin*, 35(3):40–44.

Parnas, D. L. (1985). Software Aspects of Strategic Defense Systems. *Communications of the ACM*, 28(12):1326–1335.

Parnas, D. L. and Clements, P. C. (1986). A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, 12(2):251–257.

Pashler, H., McDaniel, M., Rohrer, D., and Bjork, R. (2008). Learning Styles: Concepts and Evidence. *Psychological Science in the Public Interest*, 9(3):105–119.

Pattis, R. E. (1990). A Philosophy and Example of CS-1 Programming Projects. *SIGCSE Bulletin*, 22(1):34–39.

Pattis, R. E. (1993). The "Procedures Early" Approach in CS 1: A Heresy. *SIGCSE Bulletin*, 25(1):122–126.

Patton, M. Q. (2002). *Qualitative Research and Evaluation Methods*. Sage Publications, 3rd edition.

Paz, T. and Leron, U. (2009). The Slippery Road from Actions on Objects to Functions and Variables. *Journal for Research in Mathematics Education*, 40(1):18–39.

Pea, R. D. (1986). Language-Independent Conceptual 'Bugs' in Novice Programming. *Journal of Educational Computing Research*, 2(1):25–36.

Pears, A. and Rogalli, M. (2011). mJeliot: A Tool for Enhanced Interactivity in Programming Instruction. In: A. Korhonen and R. McCartney (eds.), *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pp. 10–15. ACM.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007). A Survey of Literature on the Teaching of Introductory Programming. *SIGCSE Bulletin*, 39(4):204–223.

Pennington, N. (1987a). Comprehension Strategies in Programming. In: G. M. Olson, S. Sheppard, and E. Soloway (eds.), *Empirical Studies of Programmers: Second Workshop*, pp. 100–113. Ablex Publishing.

Pennington, N. (1987b). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:295–341.

Pereira Mota, M., Rossy de Brito, S., Pinheiro Moreira, M., and Favero, E. L. (2009). Ambiente Integrado à Plataforma Moodle para Apoio ao Desenvolvimento das Habilidades Iniciais de Programação. In: *Anais do XX Simpósio Brasileiro de Informática na Educação*. "An Environment Integrated into the Moodle Platform for the Development of First Habits of Programming", in Portuguese.

Perkins, D. (2006). Constructivism and Troublesome Knowledge. In: J. H. F. Meyer and R. Land (eds.), *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, pp. 33–47. Routledge.

Perkins, D. (2008). Beyond Understanding. In: R. Land and J. H. F. Meyer (eds.), *Threshold Concepts within the Disciplines*, pp. 3–20. SensePublishers.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1986). Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, 2(1):37–55.

Perkins, D. N. and Martin, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In: E. Soloway and S. Iyengar (eds.), *Empirical Studies of Programmers*, pp. 213–229. Ablex Publishing.

Perkins, D. N., Schwartz, S., and Simmons, R. (1990). Instructional Strategies for the Problems of Novice Programmers. In: R. E. Meyer (ed.), *Teaching and Learning Computer Programming: Multiple Research Perspectives*, pp. 153–178. Lawrence Erlbaum.

Perlis, A. J. (1982). Epigrams on Programming. *SIGPLAN Notices*, 17(9):7–13.

Petersen, A., Craig, M., and Zingaro, D. (2011). Reviewing CS1 Exam Question Content. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pp. 631–636. ACM, New York, NY, USA.

Peterson, L. R. and Peterson, M. J. (1959). Short-Term Retention of Individual Verbal Items. *Journal of Experimental Psychology*, 58(3):193–198.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44.

Petre, M. and Green, T. R. G. (1993). Learning to Read Graphics: Some Evidence That Seeing an Information Display is an Acquired Skill. *Journal of Visual Languages & Computing*, 4(1):55–70.

Phillips, D. C. (1975). Popper and Pragmatism: A Fantasy. *Educational Theory*, 25(1):83–91.

Phillips, D. C. (1995). The Good, the Bad, and the Ugly: The Many Faces of Constructivism. *Educational Researcher*, 24(7):5–12.

Phillips, D. C. (ed.) (2000). *Constructivism in Education: Opinions and Second Opinions on Controversial Issues*. The National Society For The Study Of Education.

Phillips, D. C. (2004). Two Decades After: "After The Wake: Postpositivistic Educational Thought". *Science & Education*, 13(1):67–84.

Phillips, D. C. and Burbules, N. C. (2000). *Postpositivism and Educational Research*. Rowman & Littlefield.

Philpott, A., Robbins, P., and Whalley, J. L. (2007). Accessing The Steps on the Road to Relational Thinking. In: *20th Annual Conference of the National Advisory Committee on Computing Qualifications*, p. 286. NACCQ.

Pirolli, P. (1991). Effects of Examples and Their Explanations in a Lesson on Recursion: A Production System Analysis. *Cognition and Instruction*, 8(3):207–259.

Pirolli, P. and Recker, M. (1994). Learning Strategies and Transfer in the Domain of Programming. *Cognition and Instruction*, 12(3):235–275.

Plass, J. L., Moreno, R., and Brünken, R. (eds.) (2010). *Cognitive Load Theory*. Cambridge University Press.

Popper, K. R. (1972). *Objective Knowledge: An Evolutionary Approach*. Oxford University Press.

Posner, G. J., Strike, K. A., Hewson, P. W., and Gertzog, W. A. (1982). Accommodation of a Scientific Conception: Toward a Theory of Conceptual Change. *Science Education*, 66(2):211–227.

Pratt, C. M. (1995). *A OSF/Motif Program Animator for the DYNALAB System*. Master's thesis, Montana State University.

Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., and Carey, T. (1994). *Human Computer Interaction*. Addison-Wesley.

Price, B. A., Baecker, R. M., and Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266.

Prior McCarty, L. and Schwandt, T. A. (2000). Seductive Illusions: von Glasersfeld and Gergen on Epistemology and Education. In: D. C. Phillips (ed.), *Constructivism in Education*, pp. 41–85. The National Society For The Study Of Education.

Prosser, M. and Trigwell, K. (1999). *Understanding Teaching and Learning: The Experience in Higher Education*. Society for Research into Higher Education and Open University Press.

Proulx, V. K. (2000). Programming Patterns and Design Patterns in the Introductory Computer Science Course. *SIGCSE Bulletin*, 32(1):80–84.

Proulx, V. K. and Cashorali, T. (2005). Calculator Problem and the Design Recipe. *SIGPLAN Notices*, 40(3):4–11.

Pullen, J. (2001). The Network Workbench and Constructivism: Learning Protocols by Programming. *Computer Science Education*, 11(3):189–202.

Putnam, R. T., Sleeman, D., Baxter, J. A., and Kuspa, L. K. (1986). A Summary of Misconceptions of High School BASIC Programmers. *Journal of Educational Computing Research*, 2(4):459–72.

Ragonis, N. and Ben-Ari, M. (2005a). A Long-Term Investigation of the Comprehension of OOP Concepts by Novices. *Computer Science Education*, 15(3):203 – 221.

Ragonis, N. and Ben-Ari, M. (2005b). On Understanding the Statics and Dynamics of Object-Oriented Programs. *SIGCSE Bulletin*, 37(1):226–330.

Rajala, T., Kaila, E., and Laakso, M.-J. (n.d.). ViLLE: Collaborative Education Tool (web site). Accessed October 2011.
URL `http://ville.cs.utu.fi/`

Rajala, T., Kaila, E., Laakso, M.-J., and Salakoski, T. (2009). Effects of Collaboration in Program Visualization. In: *Technology Enhanced Learning Conference 2009*, TELearn '09.

Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. (2007). VILLE – A Language-Independent Program Visualization Tool. In: R. Lister and Simon (eds.), *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 151–159. Australian Computer Society.

Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. (2008). Effectiveness of Program Visualization: A Case Study with the ViLLE Tool. *Journal of Information Technology Education: Innovations in Practice*, 7:15–32.

Rajlich, V. (2002). Program Comprehension as a Learning Process. In: *Proceedings of the First IEEE International Conference on Cognitive Informatics*, pp. 343–347. IEEE.

Ramadhan, H. A. (2000). Programming by Discovery. *Journal of Computer Assisted Learning*, 16:83–93.

Ramadhan, H. A., Deek, F., and Shilab, K. (2001). Incorporating Software Visualization in the Design of Intelligent Diagnosis Systems for User Programming. *Artificial Intelligence Review*, 16:61–84.

Rasala, R., Raab, J., and Proulx, V. K. (2001). Java Power Tools: Model Software for Teaching Object-Oriented Design. *SIGCSE Bulletin*, 33(1):297–301.

Rasmussen, J. (1998). Constructivism and Phenomenology: What Do They Have in Common and How Can They Be Told Apart? *Cybernetics and Systems*, 29(6):553–576.

Reges, S. (2006). Back to Basics in CS1 and CS2. *SIGCSE Bulletin*, 38(1):293–297.

Renkl, A. (1997). Learning from Worked-Out Examples: A Study on Individual Differences. *Cognitive Science*, 21(1):1–29.

Renkl, A. (2002). Worked-Out Examples: Instructional Explanations Support Learning by Self-Explanations. *Learning and Instruction*, 12(5):529–556.

Renkl, A. (2005). The Worked-Out Examples Principle in Multimedia Learning. In: R. E. Mayer (ed.), *The Cambridge Handbook of Multimedia Learning*, pp. 229–246. Cambridge University Press.

Renkl, A. (2009). Why Constructivists Should Not Talk about Constructivist Learning Environments: A Commentary on Loyens and Gijbels (2008). *Instructional Science*, 37(5):495–498.

Renkl, A. and Atkinson, R. K. (2003). Structuring the Transition from Example Study to Problem Solving in Cognitive Skill Acquisition: A Cognitive Load Perspective. *Educational Psychologist*, 38(1):15–22.

Renkl, A., Stark, R., Gruber, H., and Mandl, H. (1998). Learning from Worked-Out Examples: The Effects of Example Variability and Elicited Self-Explanations. *Contemporary Educational Psychology*, 23(1):90–108.

Reynolds, C. W. and Goda, B. S. (2007). The Affective Dimension of Pervasive Themes in the Information Technology Curriculum. In: *Proceedings of the 8th ACM SIGITE conference on Information Technology Education*, SIGITE '07, pp. 13–20. ACM.

Rich, L., Perry, H., and Guzdial, M. (2004). A CS1 Course Designed to Address Interests of Women. *SIGCSE Bulletin*, 36(1):190–194.

Richards, J. (1995). Construct [ion/iv] ism: Pick One of the Above. In: L. P. Steffe and J. E. Gale (eds.), *Constructivism in Education*, pp. 57–64. Lawrence Erlbaum.

Richardson, J. T. E. (1999). The Concepts and Methods of Phenomenographic Research. *Review of Educational Research*, 69(1):53–82.

Rist, R. S. (1986). Plans in Programming: Definition, Demonstration, and Development. In: E. Soloway and S. Iyengar (eds.), *Empirical Studies of Programmers*, pp. 28–47. Ablex Publishing.

Rist, R. S. (1989). Schema Creation in Programming. *Cognitive Science*, 13:389–414.

Rist, R. S. (1995). Program Structure and Design. *Cognitive Science: A Multidisciplinary Journal*, 19(4):507–562.

Rist, R. S. (2004). Learning to Program: Schema Creation, Application, and Evaluation. *Computer Science Education Research*, pp. 175–195.

Roberts, E., Bruce, K., Cutler, R., Cross, J., Grissom, S., Klee, K., Rodger, S., Trees, F., Utting, I., and Yellin, F. (2006). ACM Java Task Force (web site). Accessed October 2011.
URL `http://jtf.acm.org`

Robinett, W. (1979). Basic Programming. Accessed October 2011.
URL `http://www.atariage.com/software_page.html?SoftwareLabelID=16`

Robins, A. (2010). Learning Edge Momentum: A New Account of Outcomes in CS1. *Computer Science Education*, 20(1):37–71.

Robins, A., Haden, P., and Garner, S. (2006). Problem Distributions in a CS1 Course. In: *Proceedings of the 8th Australian Conference on Computing Education*, ACE '06, pp. 165–173. Australian Computer Society.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172.

Roman, G.-C. and Cox, K. C. (1993). A Taxonomy of Program Visualization Systems. *Computer*, 26(12):97–123.

Ross, R. J. (1983). LOPLE: A Dynamic Library of Programming Language Examples. *SIGCUE Outlook*, 17(4):27–31.

Ross, R. J. (1991). Experience with the DYNAMOD Program Animator. *SIGCSE Bulletin*, 23(1):35–42.

Rößling, G., Mihaylov, M., and Saltmarsh, J. (2011). AnimalSense: Combining Automated Exercise Evaluations with Algorithm Animations. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pp. 298–302. ACM.

Rossy de Brito, S., Silva, A. S., de Lira Tavares, O., Favero, E. L., and Francês, C. R. L. (2011). Computer Supported Collaborative Learning for Helping Novice Students Acquire Self-Regulated Problem-Solving Skills in Computer Programming. In: H. R. Arabnia, V. A. Clincy, and L. Deligiannidis (eds.), *The 7th International Conference on Frontiers in Education: Computer Science and Computer Engineering*, FECS '11, pp. 65–73. CSREA Press.

Rountree, J. and Rountree, N. (2009). Issues Regarding Threshold Concepts in Computer Science. In: M. Hamilton and T. Clear (eds.), *Proceedings of the Eleventh Australasian Conference on Computing Education*, ACE '09, pp. 139–145. Australian Computer Society.

Rouse, W. B. and Morris, N. M. (1986). On Looking into the Black Box: Prospects and Limits in the Search for Mental Models. *Psychological Bulletin*, 100(3):349–363.

Rowbottom, D. P. (2007). Demystifying Threshold Concepts. *Journal of Philosophy of Education*, 41(2):263–270.

Rowe, G. and Thorburn, G. (2000). VINCE – An On-Line Tutorial Tool for Teaching Introductory Programming. *British Journal of Educational Technology*, 31(4):359–369.

Rubens, P. P. (ca. 1623). The Disembarkation at Marseilles (painting). In: *The Marie de' Medici Cycle*. The Louvre, Paris.

Rumelhart, D. E. (1980). Schemata: The Building Blocks of Cognition. In: W. F. Brewer, B. C. Bruce, and R. J. Spiro (eds.), *Theoretical Issues in Reading Comprehension: Perspectives from Cognitive Psychology, Linguistics, Artificial Intelligence, and Education*, pp. 33–58. Lawrence Erlbaum.

Rumelhart, D. E. and Norman, D. A. (1978). Accretion, Tuning and Restructuring: Three Modes of Learning. In: J. W. Cotton and R. Klatzky (eds.), *Semantic Factors in Cognition*, pp. 37–53. Lawrence Erlbaum.

Rumelhart, D. E. and Ortony, A. (1977). The Representation of Knowledge in Memory. In: R. C. Anderson, R. J. Spiro, and W. E. Montague (eds.), *Schooling and the acquisition of knowledge*, pp. 99–135. Lawrence Erlbaum.

Run-D.M.C. (1984). It's Like That (song). In: *Run-D.M.C.* Profile Records.

Sahlberg, P. (1996). Kuka auttaisi opettajaa – Post-moderni näkökulma opetuksen muutokseen. Julkaisusarja A 119, Kasvatustieteiden tutkimuslaitos, Jyväskylä. "Who Would Help a Teacher? A Post-Modern Perspective on Change in Teaching", in Finnish.

Sajaniemi, J. (2002). An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In: *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pp. 37–39. IEEE.

Sajaniemi, J. (n.d.). The Roles of Variables Home Page. Accessed October 2011.
URL `http://cs.joensuu.fi/~saja/var_roles/`

Sajaniemi, J., Byckling, P., and Gerdt, P. (2007). Animation Metaphors for Object-Oriented Concepts. *Electronic Notes in Theoretical Computer Science*, 178:15–22.

Sajaniemi, J. and Kuittinen, M. (2003). Program Animation Based on the Roles of Variables. In: *Proceedings of the 2003 ACM Symposium on Software visualization*, SoftVis '03, pp. 7–16. ACM.

Sajaniemi, J. and Kuittinen, M. (2005). An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education*, 15(1):59–82.

Sajaniemi, J. and Kuittinen, M. (2008). From Procedures to Objects: A Research Agenda for the Psychology of Object-Oriented Programming in Education. *Human Technology*, 4(1):75–91.

Sajaniemi, J., Kuittinen, M., and Tikansalo, T. (2008). A Study of the Development of Students' Visualizations of Program State during an Elementary Object-Oriented Programming Course. *Journal of Educational Resources in Computing*, 7(4):1–31.

Sajaniemi, J. and Navarro Prieto, R. (2005). Roles of Variables in Experts' Programming Knowledge. In: P. Romero, J. Good, E. Acosta-Chaparro, and S. Bryant (eds.), *Proceedings of the 17th Workshop of the Psychology of Programming Interest Group*, PPIG'05, pp. 145–159. PPIG.

Sajaniemi, J. and Niemeläinen, A. (1989). Program Editing Based on Variable Plans: A Cognitive Approach to Program Manipulation. In: G. Salvendy and M. J. Smith (eds.), *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, pp. 66–73. Elsevier.

Samurçay, R. (1989). The Concept of Variable in Programming: Its Meaning and Use in Problem-Solving by Novice Programmers. In: R. E. Mayer (ed.), *Studying the Novice Programmer*, pp. 161–178. Lawrence Erlbaum Associates.

Sandberg, J. (1997). Are Phenomenographic Results Reliable? *Higher Education Research & Development*, 16(2):203–212.

Sanders, I., Galpin, V., and Götschi, T. (2006). Mental Models of Recursion Revisited. *SIGCSE Bulletin*, 38(3):138–142.

Santos, A. L. (2011). AGUIA/J: A Tool for Interactive Experimentation of Objects. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pp. 43–47. ACM.

Savery, J. R. and Duffy, T. M. (1995). Problem Based Learning: An Instructional Model and Its Constructivist Framework. In: B. Wilson (ed.), *Constructivist Learning Environments: Case Studies in Instructional Design*, pp. 135–150. Educational Technology Publications.

Savin-Baden, M. (2006). Disjunction as a Form of Troublesome Knowledge in Problem-Based Learning. In: J. H. F. Meyer and R. Land (eds.), *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, pp. 160–172. Routledge.

Schank, R. C. and Abelson, R. (1977). *Scripts, Goals, Plans, and Understanding*. Lawrence Erlbaum.

Scheiter, K., Gerjets, P., and Catrambone, R. (2006). Making the Abstract Concrete: Visualizing Mathematical Solution Procedures. *Computers in Human Behavior*, 22(1):9–25.

Schmidt, H. G., Loyens, S. M. M., Van Gog, T., and Paas, F. (2007). Problem-Based Learning is Compatible with Human Cognitive Architecture: Commentary on Kirschner, Sweller, and Clark (2006). *Educational Psychologist*, 42(2):91–97.

Schneider, W. and Shiffrin, R. M. (1977). Controlled and Automatic Human Information Processing: I. Detection, Search, and Attention. *Psychological Review*, 84(1):1–66.

Schnotz, W. and Kürschner, C. (2007). A Reconsideration of Cognitive Load Theory. *Educational Psychology Review*, 19:469–508.

Schulte, C. (2008). Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching. In: *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pp. 149–160. ACM.

Schulte, C. and Bennedsen, J. (2006). What do Teachers Teach in Introductory Programming? In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 17–28. ACM.

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., and Paterson, J. H. (2010). An Introduction to Program Comprehension for Computer Science Educators. In: *ITiCSE 2010 Working Group Reports*, ITiCSE-WGR '10, pp. 65–86. ACM.

Schumacher, R. M. (1987). Acquisition of Mental Models. In: J. M. Flach (ed.), *Proceedings of the Fourth Annual Mid-Central Human Factors/ Ergonomics Conference*, pp. 142–148. Springer.

Schumacher, R. M. and Czerwinski, M. P. (1992). Mental Models and the Acquisition of Expert Knowledge. In: R. R. Hoffman (ed.), *The Psychology of Expertise: Cognitive Research and Empirical AI*, pp. 61–79. Springer.

Schumacher, R. M. and Gentner, D. (1988). Transfer of Training as Analogical Mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(4):592–600.

Schweickert, R. and Boruff, B. (1986). Short-Term Memory Capacity: Magic Number or Magic Spell? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 12(3):419–425.

Schwill, A. (1994). Fundamental Ideas of Computer Science. *Bulletin – European Association for Theoretical Computer Science*, 53:274–274.

Scott, A., Watkins, M., and McPhee, D. (2008). Progranimate – A Web Enabled Algorithmic Problem Solving Application. In: *International Conference on E-Learning, E-Business, Enterprise Information Systems, & E-Government*, EEE'08, pp. 498–508. CSREA Press.

Scott, T. (2003). Bloom's Taxonomy Applied to Testing in Computer Science Classes. *Journal of Computing in Small Colleges*, 19(1):267–274.

Scullion, J. (2002). *Are Deep Approaches to Learning Possible in Vocational Degree Courses in Construction?: A Phenomenological Inquiry*. EdD thesis, Open University.

Searle, J. R. (1992). *The Rediscovery of the Mind*. MIT Press.

Searle, J. R. (2002). Why I Am Not a Property Dualist. *Journal of Consciousness Studies*, 9(12):57–64.

Seppälä, O. (2004). Program State Visualization Tool for Teaching CS1. In: A. Korhonen (ed.), *Proceedings of the Third Program Visualization Workshop*, pp. 118–125. University of Warwick.

Sfard, A. (1991). On the Dual Nature of Mathematical Conceptions: Reflections on Processes and Objects as Different Sides of the Same Coin. *Educational Studies in Mathematics*, 22(1):1–36.

Shaffer, C. A., Naps, T. L., and Fouh, E. (2011). Truly Interactive Textbooks for Computer Science Education. In: G. Rößling (ed.), *Proceedings of the Sixth Program Visualization Workshop (PVW 2011)*, pp. 97–106. Technische Universität Darmstadt.

Shaffer, D., Doube, W., and Tuovinen, J. (2003). Applying Cognitive Load Theory to Computer Science Education. In: M. Petre and D. Budgen (eds.), *Proceedings of the Joint Conference EASE & PPIG 2003*, PPIG'03, pp. 333–346. PPIG.

Shanker, A. (1969). The Issues in the School Strike. Interviewed by William F. Buckley Jr. Accessed October 2011.
URL `http://hoohila.stanford.edu/firingline/programView2.php?programID=151`

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalley, J. L. (2008). Going SOLO to Assess Novice Programmers. *SIGCSE Bulletin*, 40(3):209–213.

Sherry, L. (1995). A Model Computer Simulation as an Epistemic Game. *SIGCSE Bulletin*, 27(2):59–64.

Shinners-Kennedy, D. (2008). The Everydayness of Threshold Concepts: State as an Example from Computer Science. In: R. Land and J. H. F. Meyer (eds.), *Threshold Concepts within the Disciplines*, pp. 119–128. SensePublishers.

Shipman, H. L. and Duch, B. J. (2001). Problem-Based Learning in Large and Very Large Classes. In: *The Power of Problem-Based Learning: a Practical "How To" for Teaching Undergraduate Courses in Any Discipline*, pp. 149–164. Stylus Pub.

Shneider, E. and Gladkikh, O. (2006). Designing Questioning Strategies for Information Technology Courses. In: *Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications*, pp. 243–248. NACCQ.

Shulman, L. S. (1986). Those Who Understand: Knowledge Growth in Teaching. *Educational Researcher*, 15(2):4–14.

Sien, V. Y. and Chong, D. W. K. (2011). Threshold Concepts in Object-Oriented Modelling. In: M. Brandsteidl and A. Winter (eds.), *7th Educators' Symposium @ MODELS 2011: Software Modeling in Education*, volume 2 of *Oldenburg Lecture Notes in Software Engineering*.

Simon (2009). A Note on Code-Explaining Examination Questions. In: A. Pears and C. Schulte (eds.), *The 9th Koli Calling International Conference on Computing Education Research*, Koli Calling '09, pp. 21–30.

Simon (2011). Assignment and Sequence: Why Some Students Can't Recognize a Simple Swap. In: A. Korhonen and R. McCartney (eds.), *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pp. 16–22. ACM.

Simon, Lopez, M., Sutton, K., and Clear, T. (2009). Surely We Must Learn to Read before We Learn to Write! In: M. Hamilton and T. Clear (eds.), *Proceedings of the Eleventh Australasian Conference on Computing Education*, ACE '09, pp. 165–170. Australian Computer Society.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J., and Warburton, G. (2012). Introductory Programming: Examining the Exams. In: M. de Raadt and A. Carbone (eds.), *Proceedings of the 14th Australasian Conference on Computing Education (ACE '12)*, volume 123 of *CRPIT*, pp. 61–70. Australian Computer Society.

Simons, D. J. and Chabris, C. F. (1999). Gorillas in our Midst: Sustained Inattentional Blindness for Dynamic Events. *Perception*, 28:1059–1074.

Sivula, K. (2005). *A Qualitative Case Study on the Use of Jeliot 3*. Master's thesis, Department of Computer Science, University of Joensuu.

Sleeman, D., Putnam, R. T., Baxter, J., and Kuspa, L. (1986). Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, 2(1):5–23.

Säljö, R. (1994). Minding Action: Conceiving of the World versus Participating in Cultural Practices. *Nordisk Pedagogik*, 14(2):71–80.

Smith, D. C. (1975). Pygmalion: A Creative Programming Environment. AI Memo STAN-CS-75-499, Department of Computer Science, Stanford University.

Smith, III, J. P., diSessa, A. A., and Roschelle, J. (1994). Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *Journal of the Learning Sciences*, 3(2):115–163.

Smith, P. A. and Webb, G. I. (1991). Debugging Using Partial Models. In: R. Godfrey (ed.), *Proceedings of the Fourth Australian Society for Computer in Learning in Tertiary Education Conference*, ASCILITE '91, pp. 581–590. University of Tasmania.

Smith, P. A. and Webb, G. I. (1995a). Reinforcing a Generic Computer Model for Novice Programmers. In: J. M. Pearce and A. Ellis (eds.), *Proceedings of the Seventh Australian Society for Computer in Learning in Tertiary Education Conference*, ASCILITE '95. University of Melbourne.

Smith, P. A. and Webb, G. I. (1995b). Transparency Debugging with Explanations for Novice Programmers. In: *Proceedings of the 2nd Workshop on Automated and Algorithmic Debugging*, AADEBUG '95. IRISA-CNRS.

Smith, P. A. and Webb, G. I. (2000). The Efficacy of a Low-Level Program Visualization Tool for Teaching Programming Concepts to Novice C Programmers. *Journal of Educational Computing Research*, 22(2):187–215.

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29(9):850–858.

Soloway, E., Adelson, B., and Ehrlich, K. (1988a). Knowledge and Processes in The Comprehension of Computer Programs. In: M. T. H. Chi, R. Glaser, and M. J. Farr (eds.), *The Nature of Expertise*, pp. 129–152. Lawrence Erlbaum.

Soloway, E., Bonar, J., and Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM*, 26(11):853–860.

Soloway, E. and Ehrlich, K. (1986). Empirical Studies of Programming Knowledge. In: C. Rich and R. C. Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*, pp. 507–521. Morgan Kaufmann.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1982). What Do Novices Know About Programming? In: A. Badre and B. Shneiderman (eds.), *Directions in Human–Computer Interactions*, pp. 27–54. Ablex Publishing.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., and Pinto, J. (1988b). Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, 31(11):1259–1267.

Sorva, J. (2007). Students' Understandings of Storing Objects. In: R. Lister and Simon (eds.), *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pp. 127–135. Australian Computer Society.

Sorva, J. (2008). The Same But Different — Students' Understandings of Primitive and Object Variables. In: A. Pears and L. Malmi (eds.), *The 8th Koli Calling International Conference on Computing Education Research*, Koli Calling '08, pp. 5–15. Uppsala University.

Sorva, J. (2010). Reflections on Threshold Concepts in Computer Programming and Beyond. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pp. 21–30. ACM.

Spohrer, J. C. and Soloway, E. (1986a). Alternatives to Construct-Based Programming Misconceptions. *SIGCHI Bulletin*, 17(4):183–191.

Spohrer, J. C. and Soloway, E. (1986b). Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632.

Spohrer, J. C., Soloway, E., and Pope, E. (1985). A Goal/Plan Analysis of Buggy Pascal Programs. *Human–Computer Interaction*, 1(2):163–207.

Stamouli, I. and Huggard, M. (2006). Object Oriented Programming and Program Correctness: The Students' Perspective. In: *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pp. 109–118. ACM.

Starr, C. W., Manaris, B., and Stalvey, R. H. (2008). Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science. *SIGCSE Bulletin*, 40(1):261–265.

Stasko, J. T. and Patterson, C. (1992). Understanding and Characterizing Software Visualization Systems. In: *Proceedings of the IEEE Workshop on Visual Languages*, pp. 3–10. IEEE.

Steffe, L. P. and Gale, J. E. (eds.) (1995). *Constructivism in Education*. Lawrence Erlbaum.

Stoodley, I., Christie, R., and Bruce, C. (2004). Masters Students' Experiences of Learning to Program: An Empirical Model. In: *Proceedings of the International Conference on Qualitative Research in IT & IT in Qualitative Research*, QualIT '04.

Strike, K. A. and Posner, G. J. (1985). A Conceptual Change View of Learning and Understanding. In: L. H. T. West and A. Pines (eds.), *Cognitive Structure and Conceptual Change*, pp. 211–231. Academic Press.

Stützle, T. and Sajaniemi, J. (2005). An Empirical Evaluation of Visual Metaphors in the Animation of Roles of Variables. *Informing Science Journal*, 8:87–100.

Suthers, D. D. and Hundhausen, C. D. (2003). An Experimental Study of the Effects of Representational Guidance on Collaborative Learning Processes. *Journal of the Learning Sciences*, pp. 183–218.

Sutinen, E., Tarhio, J., Lahtinen, S.-P., Tuovinen, A.-P., Rautama, E., and Meisalo, V. (1997). Eliot – An Algorithm Animation Environment. Teaching and Learning Report A-1997-4, Department of Computer Science, University of Helsinki.
URL `http://www.cs.helsinki.fi/TR/A-1997/4/A-1997-4.ps.gz`

Sweller, J. (1988). Cognitive Load During Problem Solving: Effects on Learning. *Cognitive Science*, 12(2):257–285.

Sweller, J. (2005). The Reduncancy Principle in Multimedia Learning. In: R. E. Mayer (ed.), *The Cambridge Handbook of Multimedia Learning*, pp. 159–168. Cambridge University Press.

Sweller, J. (2010a). Cognitive Load Theory: Recent Theoretical Advances. In: J. L. Plass, R. Moreno, and R. Brünken (eds.), *Cognitive Load Theory*, pp. 29–47. Cambridge University Press.

Sweller, J. (2010b). Element Interactivity and Intrinsic, Extraneous and Germane Cognitive Load. *Educational Psychology Review*, 22:123–138.

Sweller, J. and Chandler, P. (1994). Why Some Material is Difficult to Learn. *Cognition and Instruction*, 12(3):185–233.

Sweller, J., Kirschner, P. A., and Clark, R. E. (2007). Why Minimally Guided Teaching Techniques Do Not Work: A Reply to Commentaries. *Educational Psychologist*, 42(2):115–121.

Tashakkori, A. and Teddlie, C. (eds.) (2010). *Sage Handbook of Mixed Methods in Social & Behavioral Research*. Sage, 2nd edition.

Teague, D., Corney, M., Ahadi, A., and Lister, R. (2012). Swapping as the "*Hello World*" of Relational Reasoning: Replications, Reflections and Extensions. In: M. de Raadt and A. Carbone (eds.), *Proceedings of the 14th Australasian Conference on Computing Education (ACE '12)*, volume 123 of *CRPIT*, pp. 87–93. Australian Computer Society.

Tedre, M. and Sutinen, E. (2008). Three Traditions of Computing: What Educators Should Know. *Computer Science Education*, 18(3):153–170.

Teif, M. and Hazzan, O. (2006). Partonomy and Taxonomy in Object-Oriented Thinking: Junior High School Students' Perceptions of Object-Oriented Basic Concepts. *SIGCSE Bulletin*, 38(4):55–60.

Thomas, L., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., and Zander, C. (2010). Threshold Concepts in Computer Science: An Ongoing Empirical Investigation. In: J. H. F. Meyer, R. Land, and C. Baillie (eds.), *Threshold Concepts and Transformational Learning*, pp. 241–258. SensePublishers.

Thomas, L., Ratcliffe, M., and Thomasson, B. (2004). Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results. *SIGCSE Bulletin*, 36(1):250–254.

Thompson, E. (2008). *How Do They Understand? Practitioner Perceptions of an Object-Oriented Program*. Doctoral thesis, Massey University.

Thompson, E. (2010). From Phenomenography Study to Planning Teaching. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pp. 13–17. ACM.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., and Robbins, P. (2008). Bloom's Taxonomy for CS Assessment. In: *Proceedings of the Tenth Conference on Australasian Computing Education*, ACE '08, pp. 155–161. Australian Computer Society.

Thota, N., Berglund, A., and Clear, T. (2012). Illustration of Paradigm Pluralism in Computing Education Research. In: M. de Raadt and A. Carbone (eds.), *Proceedings of the 14th Australasian Conference on Computing Education (ACE '12)*, volume 123 of *CRPIT*, pp. 103–112. Australian Computer Society.

Thota, N. and Whitfield, R. (2010). Holistic Approach to Learning and Teaching Introductory Object-Oriented Programming. *Computer Science Education*, 20(2):103–127.

Thuné, M. and Eckerdal, A. (2009). Variation Theory Applied to Students' Conceptions of Computer Programming. *European Journal of Engineering Education*, 34(4):339–347.

Thuné, M. and Eckerdal, A. (2010). Students' Conceptions of Computer Programming. Technical Report 2010-021, Department of Information Technology, Uppsala University.

Tobias, S. and Duffy, T. M. (2009). *Constructivist Instruction: Success or Failure?* Taylor & Francis.

Tognazzini, B. (2003). First Principles of Interaction Design. Accessed October 2011. URL `http://www.asktog.com/basics/firstPrinciples.html`

Tuovinen, J. E. (2000). Optimising Student Cognitive Load in Computer Education. In: *Proceedings of the Australasian Conference on Computing Education*, ACE '00, pp. 235–241. ACM.

Turner, J. M. and Bélanger, F. P. (1996). Escaping from Babel: Improving the Terminology of Mental Models in the Literature of Human–Computer Interaction. *Canadian Journal of Information and Library Science*, 21(3/4):35–58.

Uljens, M. (1996). On the Philosophical Foundation of Phenomenography. In: G. Dall'Alba and B. Hasselgren (eds.), *Reflections on Phenomenography – Toward a Methodology?*, pp. 105–130. University of Gothenburg.

Uljens, M. (1998). Fenomenografin, dess icke-dualistiska ontologi och Menons paradox. *Pedagogisk Forskning i Sverige*, 3(2):122–129. "Phenomenography, its non-dualist ontology, and Meno's paradox", in Swedish.

Urquiza-Fuentes, J. and Velázquez-Iturbide, J. Á. (2007). An Evaluation of the Effortless Approach to Build Algorithm Animations with WinHIPE. *Electronic Notes in Theoretical Computer Science*, 178:3–13.

Urquiza-Fuentes, J. and Velázquez-Iturbide, J. Á. (2009). A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems. *ACM Transactions on Computing Education*, 9(2):1–21.

Uttal, W. R. (2000). *The War between Mentalism and Behaviorism: On the Accessibility of Mental Processes*. Lawrence Erlbaum.

Uttal, W. R. (2004). *Dualism: The Original Sin of Cognitivism*. Lawrence Erlbaum.

Vagianou, E. (2006). Program Working Storage: A Beginner's Model. In: A. Berglund and M. Wiggberg (eds.), *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, Koli Calling '06, pp. 69–76. Uppsala University.

Vainio, V. (2006). *Opiskelijoiden mentaaliset mallit ohjelmien suorituksesta ohjelmoinnin peruskurssilla*. Master's thesis, Department of Psychology, University of Helsinki. "Students' Mental Models of Program Execution in an Elementary Programming Course", in Finnish.

Vainio, V. and Sajaniemi, J. (2007). Factors in Novice Programmers' Poor Tracing Skills. *SIGCSE Bulletin*, 39(3):236–240.

Valentine, D. W. (2004). CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. *SIGCSE Bulletin*, 36(1):255–259.

van Dijk, T. A. and Kintsch, W. (1983). *Strategies of Discourse Comprehension*. Academic Press.

Van Gorp, M. J. and Grissom, S. (2001). An Empirical Evaluation of Using Constructive Classroom Activities to Teach Introductory Programming. *Computer Science Education*, 11(3):247–260.

van Merriënboer, E. and Krammer, H. P. M. (1987). Instructional Strategies and Tactics for the Design of Introductory Computer Programming Courses in High School. *Instructional Science*, 16(3):251–285.

van Merriënboer, J. J. G. (1990). Strategies for Programming Instruction in High School: Program Completion vs. Program Generation. *Journal of Educational Computing Research*, 6(3):265–285.

van Merriënboer, J. J. G. and de Croock, M. B. M. (1992). Strategies for Computer-Based Programming Instruction: Program Completion vs. Program Generation. *Journal of Educational Computing Research*, 8(3):365–394.

van Merriënboer, J. J. G. and Kirschner, P. A. (2007). *Ten Steps to Complex Learning: A Systematic Approach to Four-Component Instructional Design*. Lawrence Erlbaum.

van Merriënboer, J. J. G., Kirschner, P. A., and Kester, L. (2003). Taking the Load Off a Learner's Mind: Instructional Design for Complex Learning. *Educational Psychologist*, 38(1):5–13.

VanLehn, K. (1996). Cognitive Skill Acquisition. *Annual Review of Psychology*, 47:513–539.

Velázquez-Iturbide, J. Á., Pérez-Carrasco, A., and Urquiza-Fuentes, J. (2008). SRec: An Animation System of Recursion for Algorithm Courses. *SIGCSE Bulletin*, 40(3):225–229.

Venables, A., Tan, G., and Lister, R. (2009). A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In: *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pp. 117–128. ACM.

Victor, B. (2012). Inventing on Principle (video). Accessed February 2012.
URL `http://vimeo.com/36579366`

Virtanen, A. T., Lahtinen, E., and Järvinen, H.-M. (2005). VIP, a Visual Interpreter for Learning Introductory Programming with C++. In: T. Salakoski, T. Mäntylä, and M. Laakso (eds.), *Proceedings of the Fifth Koli Calling Conference on Computer Science Education*, Koli Calling '05, pp. 125–130. Turku Centre for Computer Science.

Visser, W. (1987). Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer. In: G. M. Olson, S. Sheppard, and E. Soloway (eds.), *Empirical Studies of Programmers: Second Workshop*, pp. 217–230. Ablex Publishing.

Visser, W. and Hoc, J.-M. (1990). Expert Software Design Strategies. In: J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore (eds.), *Psychology of Programming*, pp. 235–250. Academic Press.

von Glasersfeld, E. (1982). An Interpretation of Piaget's Constructivism. *Revue Internationale de Philosophie*, 36(4):612–635.

von Glasersfeld, E. (1995). A Constructivist Approach to Teaching. In: L. P. Steffe and J. E. Gale (eds.), *Constructivism in Education*. Lawrence Erlbaum.

von Glasersfeld, E. (1998). Why Constructivism Must Be Radical. In: M. Larochelle, N. Bednarz, and J. Garrison (eds.), *Constructivism and Education*, pp. 23–28. Cambridge University Press.

von Mayrhauser, A. and Vans, A. M. (1995). Program Understanding: Models and Experiments. In: M. Yovits and M. Zelkovitz (eds.), *Advances in Computers*, volume 40, pp. 1–37. Academic Press.

Walker, H. M. (2011). Resolved: Ban 'Programming' from Introductory Computing Courses. *ACM Inroads*, 2(4):16–17.

Wallingford, E. (1996). Toward a First Course Based on Object-Oriented Patterns. *SIGCSE Bulletin*, 28(1):27–31.

Walsh, E. (2000). Phenomenographic Analysis of Interview Transcripts. In: J. A. Bowden and E. Walsh (eds.), *Phenomenography*, pp. 19–33. RMIT University Press.

Webb, G. (1997). Deconstructing Deep and Surface: Towards a Critique of Phenomenography. *Higher Education*, 33(2):195–212.

Weber, G. and Brusilovsky, P. (2001). ELM-ART: An Adaptive Versatile System for Web-Based Instruction. *International Journal of Artificial Intelligence in Education*, 12:351–384.

Weigend, M. (n.d.). The Python Visual Sandbox (web site). Accessed October 2011. URL `http://www.python-visual-sandbox.de/`

Wesley, J. J. (2011). Observing the Political World: Quantitative and Qualitative Approaches. In: K. Archer and L. Berdahl (eds.), *Explorations: Conducting Empirical Research in Canadian Political Science*, pp. 123–144. Oxford University Press, 2nd edition.

Westbrook, L. (2006). Mental Models: A Theoretical Overview and Preliminary Study. *Journal of Information Science*, 32(6):563–579.

Westfall, R. (2001). Hello, World Considered Harmful. *Communications of the ACM*, 44(10):129–130.

Whalley, J., Clear, T., Robbins, P., and Thompson, E. (2011). Salient Elements in Novice Solutions to Code Writing Problems. *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, 114:35–44.

Whalley, J. L., Clear, T., and Lister, R. (2007). The Many Ways of the BRACElet Project. *Bulletin of Applied Computing and Information Technology*, 5(1).

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., and Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In: *Proceedings of the 8th Australasian Conference on Computing Education*, ACE '06, pp. 243–252. Australian Computer Society.

Wickens, C. D. (1996). *Engineering Psychology and Human Performance*. Schools, 2nd edition.

Wiedenbeck, S. (1989). Learning Iteration and Recursion from Examples. *International Journal of Man–Machine Studies*, 30(1):1–22.

Wiedenbeck, S., Fix, V., and Scholtz, J. (1993). Characteristics of the Mental Representations of Novice and Expert Programmers: an Empirical Study. *International Journal of Man–Machine Studies*, 39(5):793–812.

Wiedenbeck, S. and Ramalingam, V. (1999). Novice Comprehension of Small Programs Written in the Procedural and Object-Oriented Styles. *International Journal of Human–Computer Studies*, 51(1):71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. L. (1999). A Comparison of the Comprehension of Object-Oriented and Procedural Programs by Novice Programmers. *Interacting with Computers*, 11(3):255–282.

Willingham, D. T. (2009). *Why Don't Students Like School? A Cognitive Scientist Answers Questions about How the Mind Works and What it Means for Your Classroom*. Jossey-Bass.

Winslow, L. E. (1996). Programming Pedagogy – A Psychological Overview. *SIGCSE Bulletin*, 28(3):17–22.

Wittwer, J. and Renkl, A. (2010). How Effective are Instructional Explanations in Example-Based Learning? A Meta-Analytic Review. *Educational Psychology Review*, 22:393–409.

Wolfman, S. A. (2002). Making Lemonade: Exploring the Bright Side of Large Lecture Classes. *SIGCSE Bulletin*, 34(1):257–261.

Wulf, T. (2005). Constructivist Approaches for Teaching Computer Programming. In: *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, pp. 245–248. ACM.

Yadin, A. (2011). Reducing the Dropout Rate in an Introductory Programming Course. *ACM Inroads*, 2(4):71–76.

Yehezkel, C., Ben-Ari, M., and Dreyfus, T. (2007). The Contribution of Visualization to Learning Computer Architecture. *Computer Science Education*, 17(2):117 – 127.

Zander, C., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Sanders, K. (2008). Threshold Concepts in Computer Science: A Multi-National Empirical Investigation. In: R. Land and J. H. F. Meyer (eds.), *Threshold Concepts within the Disciplines*, pp. 105–118. SensePublishers.

Zander, C., Boustedt, J., McCartney, R., Moström, J. E., Sanders, K., and Thomas, L. (2009). Student Transformations: Are They Computer Scientists Yet? In: *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pp. 129–140. ACM.

Zendler, A., Spannagel, C., and Klaudt, D. (2008). Process as Content in Computer Science Education: Empirical Determination of Central Processes. *Computer Science Education*, 18(4):231–245.

Aalto-DD 61/2012

ISBN 978-952-60-4625-9
ISBN 978-952-60-4626-6 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS +
ECONOMY

ART +
DESIGN +
ARCHITECTURE

SCIENCE +
TECHNOLOGY

CROSSOVER

DOCTORAL
DISSERTATIONS