Department of Computer Science and Engineering

Supporting Acquisition of Programming Skills in Introductory Programming Education

Environments for Practicing Programming and Recording and Analysis of Exercise Sessions

Juha Helminen

Aalto University

DOCTORAL
DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 69/2014

Supporting Acquisition of Programming Skills in Introductory Programming Education

Environments for Practicing Programming and Recording and Analysis of Exercise Sessions

Juha Helminen

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 6 June 2014 at 12.

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

Supervising professor
Professor Lauri Malmi

Thesis advisors
D.Sc. (Tech) Petri Ihantola
D.Sc. (Tech) Ville Karavirta

Preliminary examiners
Professor Mordechai Ben-Ari, Weizmann Institute of Science, Israel
Associate Professor Mike Joy, University of Warwick, UK

Opponent
Professor Peter Brusilovsky, University of Pittsburgh, USA

Aalto University publication series
DOCTORAL DISSERTATIONS 69/2014

ISBN 978-952-60-5687-6
ISBN 978-952-60-5688-3 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-5688-3

Unigrafia Oy
Helsinki 2014

Finland

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

Abstract

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

Author
Juha Helminen

Name of the doctoral dissertation
Supporting Acquisition of Programming Skills in Introductory Programming Education: Environments for Practicing Programming and Recording and Analysis of Exercise Sessions

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 69/2014

Field of research Software Systems

Manuscript submitted 11 February 2014 **Date of the defence** 6 June 2014

Permission to publish granted (date) 23 April 2014 **Language** English

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

Abstract

The work in this thesis falls under two themes. First, we have experimented with a number of novel tools to lower the barrier to start practicing programming skills. Specifically, we present experiences on using Python novice environments that reduce the complexity of getting started with practicing programming in the following two ways. First, only a limited set of key functionality is provided in an integrated exercise environment. Second, only web technologies are used to improve portability and ease of access. Additionally, we present experiences on using a special type of program construction exercises, as well as, improvements to the automated feedback provided in these exercises. Finally, we present an application for practicing Python programming on mobile touch devices that is based on these exercises.

As for the second theme, we have carried out automated recording of students' exercise sessions and explored what can be learned from such data. Particularly, we show how to visualize program construction exercise sessions as a graph in order to reveal common patterns and anomalies. We identified two overall patterns of constructing programs: line-by-line and control structures first. We also identified behavior that seems to be indicative of difficulties: backtracking, going in circles, and excessive, trial-and-error use of feedback.

Additionally, we use this type of data to evaluate the effect of different types of feedback in program construction exercises. Students who received execution-based feedback needed on average more steps and took longer to solve an exercise than those who got line-based feedback. On the other hand, execution-based feedback was requested less frequently and the respective code was more commonly executable.

Finally, we make use of automatically recorded data on programming sessions to identify and quantify how students use an interactive Python console, as well as, to study how frequently and which kinds of execution errors they encounter. Students made use of the console both for testing their code and for exploring language features. A variety of error types were observed while only a minority of those accounted for the majority of occurrences.

As the key results of this thesis, many of the studied approaches to supporting the acquisition of programming skills have been successfully used on programming courses together with automated recording of exercise sessions that, in turn, has been made use of to identify and quantify common patterns and difficulties for the benefit of teaching and education research.

Keywords introductory programming education, automated assessment, program visualization, novice programming environment, program construction exercise, mobile learning, programming process, programming session

ISBN (printed) 978-952-60-5687-6 **ISBN (pdf)** 978-952-60-5688-3

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

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2014

Pages 180 **urn** http://urn.fi/URN:ISBN:978-952-60-5688-3

A? Aalto-yliopisto

Tiivistelmä

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

Tekijä
Juha Helminen

Väitöskirjan nimi
Ohjelmointitaitojen kehittämisen tukeminen ohjelmoinnin perusopetuksessa: ohjelmoinnin harjoitteluympäristöt ja työskentelyn tallentaminen ja analysointi

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 69/2014

Tutkimusala Ohjelmistojärjestelmät

Käsikirjoituksen pvm 11.02.2014 **Väitöspäivä** 06.06.2014

Julkaisuluvan myöntämispäivä 23.04.2014 **Kieli** Englanti

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

Tiivistelmä

Väitöskirjan työ jakautuu kahden teeman alle. Ensinnäkin työssä on tutkittu ohjelmoinnin harjoittelun aloittamiskynnyksen madaltamista uudenlaisilla ohjelmatyökaluilla. Työssä esitetään kokemuksia sellaisista aloittelijan Python-ympäristöistä, jotka vähentävät ohjelmoinnin harjoittelun aloittamisen mutkikkuutta seuraavalla kahdella tavalla. Ensinnäkin näissä tarjotaan vain rajallinen määrä avaintoimintoja integroidussa harjoitusympäristössä. Toisekseen näissä käytetään vain web-teknologioita siirrettävyyden ja käytön helpottamiseksi. Lisäksi työssä esitetään kokemuksia uudenlaisten ohjelman rakennustehtävien opetuskäytöstä sekä parannuksia näissä annettavaan automaattiseen palautteeseen. Työssä esitetään myös näihin tehtäviin perustuva sovellus Python-ohjelmoinnin harjoitteluun kosketusnäytöllisillä mobiililaitteilla.

Opiskelijoiden työskentelyä on myös tallennettu automatisoidusti ja tutkittu mitä tällaisesta datasta voi oppia. Työssä näytetään kuinka rakennustehtävien ratkomista voidaan havainnollistaa verkkona, jotta saadaan näkyville yleisiä malleja ja poikkeavuuksia. Työssä tunnistetaan kaksi yleismallia rakentamiselle: rivi kerrallaan ja kontrollirakenteet ensin. Myös vaikeuksiin viittaavaa käyttäytymistä havaittiin: peruuttaminen, ympyrää kiertäminen ja liiallinen, yritys ja erehdys -tyyppinen palautteen käyttö.

Tämäntyyppistä dataa käytetään myös ohjelman rakennustehtävien erityyppisten palautteiden vaikutuksen arviointiin. Opiskelijat, jotka saivat suorituspohjaista palautetta, tarvitsivat keskimäärin enemmän askeleita ja käyttivät enemmän aikaa tehtävän ratkaisemiseen kuin ne, jotka saivat rivipohjaista palautetta. Toisaalta suorituspohjaista palautetta pyydettiin harvemmin ja vastaava ohjelmakoodi oli useammin suoritettavissa.

Ohjelmointiharjoittelusta automatisoidusti tallennettua dataa hyödynnettiin sen tunnistamisessa ja kvantifioinnissa kuinka opiskelijat käyttävät interaktiivista Python-konsolia sekä suoritusvirheiden tutkimisessa. Opiskelijat hyödynsivät konsolia sekä koodinsa testaamiseen että kielen ominaisuuksien tutkimiseen. Vaihtelevia virhetyyppejä havaittiin, mutta pieni joukko yleisimpiä virhetyyppejä kattoi enemmistön esiintymistä.

Väitöskirjan päätulos on, että monia tutkittuja lähestymistapoja ohjelmointitaitojen kehittämisen tukemiseen on onnistuneesti käytetty ohjelmointikursseilla yhdessä automatisoidun työskentelyn tallentamisen kanssa, jota vuorostaan on hyödynnetty yleisten mallien ja vaikeuksien tunnistamisessa sekä kvantifioinnissa opetuksen ja opetustutkimuksen edistämiseksi.

Avainsanat ohjelmoinnin perusopetus, automaattinen arviointi, ohjelmavisualisaatio, aloittelijan ohjelmointiympäristö, ohjelman rakennustehtävä, mobiilioppiminen, ohjelmointiprosessi, ohjelmointisessio

ISBN (painettu) 978-952-60-5687-6 **ISBN (pdf)** 978-952-60-5688-3

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

Julkaisupaikka Helsinki **Painopaikka** Helsinki **Vuosi** 2014

Sivumäärä 180 **urn** http://urn.fi/URN:ISBN:978-952-60-5688-3

Preface

All the way back in May 2006, Ari Korhonen hired me to work as a research assistant at the Helsinki University of Technology. I joined the Software Visualization Group led by Ari and was tasked with a programming job dealing with Jussi Nikander's doctoral research. I had no idea what I was getting into. I ended up co-authoring several publications about this work with Jussi and Ari. This was my introduction to scientific research and eventually led me down the path of doing a doctoral thesis of my own. Thank you, Jussi and Ari, for showing me the ropes.

In 2008, I began with my own research as I started working on my master's thesis under the supervision of Professor Lauri Malmi. This work eventually led to the first publication included in this doctoral thesis. After completing my master's thesis, Lauri continued to supervise my doctoral studies which I began in 2009. Thank you, Lauri, for providing me with this great opportunity and for your patience to see it through. Later on, Petri Ihantola and Ville Karavirta took on the responsibilities of being my thesis advisors. Above all, I thank you, Lauri, Petri, and Ville, for your continuous guidance and support during this process. You have been instrumental in keeping me on track and I would like to express my deepest gratitude to you for helping me to find the inspiration, to acquire the skills, and to build up the knowledge base required to finally reach this goal.

Overall, several people have contributed to the research reported in this thesis, both directly and indirectly. I thank my co-authors, Lauri, Petri, Ville, and Satu Alaoutinen, for the fruitful collaboration. I also thank Petri, Ville, Satu, and Kerttu Pollari-Malmi for letting me experiment with new educational technologies on their programming courses. I also thank my colleagues, all the current and former members of the Learning+Technology research group, for the many stimulating discussions about programming, education, and research. Special thanks go to long-time members Otto Seppälä and Juha Sorva for their many insights about programming education. I also extend my thanks to Lasse Hakulinen and Tapio Auvinen, my fellow travellers on the journey to a doctoral degree. It has been a privilege to have you as my cubicle mates and I hope to see you make it to the finish line soon too.

Furthermore, I thank the pre-examiners, Professor Mordechai Ben-Ari and Associate Professor Mike Joy, for taking the time to read my thesis and for their valuable comments and suggestions on how to improve this work. I am also honored to have Professor Peter Brusilovsky as my opponent.

Finally, I thank my family for their tireless support and constant encouragement over all these years and throughout my life, and my friends for keeping me sane all the way through this long and arduous process.

Espoo, May 8, 2014,

Juha Helminen

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Juha Helminen and Lauri Malmi. Jype – A Program Visualization and Programming Exercise Tool for Python. In *Proceedings of the ACM Symposium on Software Visualization (SOFTVIS'10)*, Salt Lake City, Utah, USA, pages 153–162, October 2010.

II Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. How Do Students Solve Parsons Programming Problems? – An Analysis of Interaction Traces. In *Proceedings of the Eighth Annual International Computing Education Research Conference (ICER '12)*, Auckland, New Zealand, pages 119–126, September 2012.

III Ville Karavirta, Juha Helminen, and Petri Ihantola. A Mobile Learning Application for Parsons Problems with Automatic Feedback. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling '12)*, Tahko, Finland, pages 11–18, November 2012.

IV Juha Helminen, Petri Ihantola, Ville Karavirta, and Satu Alaoutinen. How Do Students Solve Parsons Programming Problems? – Execution-Based vs. Line-Based Feedback. In *Proceedings of the International Conference on Learning and Teaching in Computing and Engineering (LaTiCE '13)*, Macau, China, pages 55–61, March 2013.

V Juha Helminen, Petri Ihantola, and Ville Karavirta. Recording and Analyzing In-Browser Programming Sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education*

Research (Koli Calling '13), Koli, Finland, pages 13–22, November 2013.

VI Petri Ihantola, Juha Helminen, and Ville Karavirta. How to Study Programming on Mobile Touch Devices – Interactive Python Code Exercises. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)*, Koli, Finland, pages 51–58, November 2013.

Author's Contribution

Publication I: "Jype – A Program Visualization and Programming Exercise Tool for Python"

Publication I describes the rationale for and the design and implementation of a program visualization and programming exercise tool for Python.

The author of this thesis is the corresponding author of this paper and led the study. He designed and implemented the tool. The other author contributed to reporting.

Publication II: "How Do Students Solve Parsons Programming Problems? – An Analysis of Interaction Traces"

Publication II presents visualizations and analyses on students' program construction sessions in the js-parsons environment based on recorded interaction traces.

The author of this thesis is the corresponding author of this paper and led the study. He designed and implemented the visualizations and the analyses. He was responsible for the analysis of data and the reporting of results but the other authors also contributed to these.

Publication III: "A Mobile Learning Application for Parsons Problems with Automatic Feedback"

Publication III describes the rationale for and the design and implementation of new feedback mechanisms in program construction exercises in the js-parsons environment and program construction exercises on mobile touch devices.

The author of this thesis designed the algorithms for all the new feedback mechanisms and carried out analyses on the expected effects. He was also responsible for the Android implementation of the environment. The corresponding author was responsible for the iOS implementation of the environment. All the authors contributed evenly to other parts of the paper.

Publication IV: “How Do Students Solve Parsons Programming Problems? – Execution-Based vs. Line-Based Feedback”

Publication IV describes the rationale for and the design and implementation of a new type of execution-based feedback on program construction exercises in the js-parsons environment and presents the results of an experiment to study how the type of feedback affects how students solve these exercises.

The author of this thesis is the corresponding author of this paper and led the study. He designed and implemented the analyses. He was responsible for the analysis of data and the reporting of results but the other authors also contributed to these.

Publication V: “Recording and Analyzing In-Browser Programming Sessions”

Publication V describes the implementation of a browser-based programming environment and presents observations from the analysis of programming sessions recorded using this environment.

The author of this thesis is the corresponding author of this paper and led the study. He designed and implemented the programming environment and the analyses. He was responsible for the analysis of data and the reporting of results but the other authors also contributed to these.

Publication VI: "How to Study Programming on Mobile Touch Devices – Interactive Python Code Exercises"

Publication VI describes approaches to practicing programming on mobile devices and the rationale for and the design and implementation of a new type of program construction exercise for mobile touch devices.

All the authors contributed evenly to all parts of this paper.

1. Introduction

1.1 Demand for Programmers

Considering our growing reliance on information technologies both as a society and as individuals we can tell that the demand for expertise in the art of computer science is anything but going down. Consequently, computer science education plays a key role in building and sustaining the infrastructure of our information society and we must strive to design and utilize best possible methods and tools in the learning and teaching of computer science in order to train true professionals for the needs of the future.

Computer science degree programs invariably begin with courses on introductory programming. That is, after all, the cornerstone of computing – by definition, a computer is a programmable device. On the other hand, due to the pervasiveness of information technologies, competency in computing has become highly relevant in many other fields, as well, and, today, the basics of programming are being taught to a wide array of engineering students enrolled in a variety of different majors. This, in part, has resulted in a notable increase in the number of students learning to program and, at many universities, such as ours [72], introductory programming courses have hundreds of students. Outside of engineering disciplines, knowledge of programming may still benefit individuals as simply a different mode of thought and as preparation for interacting with technology in everyday life, as suggested by Kelleher and Pausch [70].

1.2 Learning to Program Is Hard

Unfortunately, many find learning to program very challenging. Simon has reviewed a number of studies that have investigated the abilities of first-year programming students and provide evidence of this [126]. Often cited are the findings of the 2001 ITiCSE working group led by McCracken. A sample of 216 students from 4 universities were assessed after taking their first CS courses. They were given a time-restricted lab-based programming task where on average they managed to score a disappointing 22.89 points out of 110 according to the evaluation criteria [92].

Furthermore, anecdotally, programming courses have very high dropout rates [117]. Bennedsen and Caspersen carried out an international[1] survey to verify and quantify this phenomenon and found that, among the 63 institutions that answered, on average 72 % of students passed the first programming course, commonly referred to as CS1, while there was a huge variation in the pass rates among the different institutions [16]. They also noted that it seemed small classes did better on average in this regard.

1.3 Challenges in Learning and Teaching Programming

What makes learning to program particularly challenging is the multitude of knowledge and skills one must acquire simultaneously. You need to grasp many abstract concepts and constructs, such as variables, functions, objects, algorithms, and data structures; familiarize yourself with several tools, such as design tools, compilers, and debuggers; and learn systematic approaches to, and strategies and processes for using these skills and knowledge in analyzing a problem, in some perhaps entirely different domain, in order to develop and implement an appropriate software solution. Du Boulay lists five areas of difficulty in learning to program: (1) general orientation, what programs are and what to do with them; (2) the notional machine, a model of the computer as defined by the execution environment; (3) notation, the syntax and semantics of programming languages; (4) structures, programming patterns, commonly referred to as plans or schemas; and (5) pragmatics, as in, actually planning, implementing, testing, and debugging programs using appropriate methods and tools. He goes on to state that "none of these issues are entirely separable from the others, and much of the shock . . . of the first few encounters between the

[1] Two thirds of answers originated from the US.

learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once." [39]

Large classes with insufficient resources for adequate individual guidance then only exacerbate the problems. Further source of difficulties is the heterogeneity of the course participants that makes it challenging to design teaching in such a way that it addresses everyone's needs. Indeed, many students have previous experience and thus some established notions of programming, be they correct or not, and if there are students from different majors, the motivations and goals of the students will vary greatly, as well. For example, Lahtinen et al. carried out an international survey to map out the difficulties experienced and perceived by students and teachers of introductory programming classes. In total, 559 students from 6 different universities filled out the survey. More than half of the students had previous experience in programming prior to their university studies [82].

As learning to program is typically the first hurdle to cross in a computer science degree, it carries special meaning there. On the one hand, poor understanding of basic concepts and processes will surely make more advanced studies troublesome. On the other hand, if the introductory programming courses are generally regarded as overly difficult with a higher than usual failure rate, this is sure to discourage students from entering the discipline. This is especially worrisome due to the pervasiveness of software, and computing in general. Because software is so ubiquitous, there is a certain need for having all kinds of people involved in designing and implementing computing solutions. In discussing the importance of lowering the barriers to programming for all people, Kelleher and Pausch note the same idea that if the population of people creating software more closely matches the population using software, the software designed will probably better match users' needs [70]. Furthermore, from a practical point of view it is a huge waste of resources, on both parts – for both the students and the teachers – to have students take up a course and then fail or drop out in the middle.

1.4 Scope of this Thesis

The work in this thesis is situated in the field of computing education research and, more specifically, the learning and teaching of programming. This area of study lies in the general context of cognitive psychology and has ties to such topics as, for example, problem solving and knowledge representation. In this thesis, we are particularly interested in how to study and support learning in the context of large courses where there is a lack of resources or opportunities for face-to-face individual guidance and there is also an emphasis on enabling distance learning. Within this context, there are still various ways to support the process of learning to program, such as, for example, aiming to facilitate learning from peers. However, in this work, we have studied supporting acquisition of programming skills with software tools that, in particular, aim to facilitate independent practice.

Furthermore, we have focused on adult learners with an immediate need of learning to program for future professional careers. Thus, as opposed to studies of how to introduce programming concepts to children, in our context, it is essential to aim for a level of proficiency that also goes beyond abstraction and into practice. Accordingly, the empirical data collected in this thesis comes from university-level students.

Specifically, the first overall research theme in this thesis is the goal of minimizing the barrier to start practicing programming. We have approached this in a number of ways that are discussed in more detail in the next section. To summarize, we have designed, implemented, and experimented with environments for practicing programming that combine and develop ideas from several fields of programming education tools research: program visualization, automated assessment and feedback, and programming environments for novices. The tools have been developed to support teaching Python at our university. Consequently, our work relates better to studies with this same focus than those dealing with some other programming language.

The second overall research theme in this work is the recording and analysis of learners' exercise sessions. We have used some of the tools developed to collect data on how learners work and explored the use of this data in investigating their actions and behavior in order to inform teaching and computing education research. This area of study will also be discussed further in the next section.

Overall, the work in this thesis has a significant constructive component that draws upon advances in software technology. In particular, we have studied the design and implementation of web-based programming environments for novices, automated feedback in program construction exercises, exercises for programming on mobile touch devices, and the automated recording of exercise sessions. This technical perspective means that the details of how learning occurs and of how the tools impact learning are mostly out of scope. Having said that, the basic theoretical foundations of learning to program will be discussed in the next chapter. All in all, we have taken an empirical approach to gaining knowledge. We have experimented with different learning technologies on programming courses and collected data via questionnaires and more importantly by recording exercise sessions. We have then analyzed this data both qualitatively and quantitatively mostly by making use of various means of visualization and statistical methods.

1.5 Research Questions

The work in this thesis falls under two distinct themes which are discussed below separately.

1.5.1 Environments for Practicing Programming

Right from the outset, novice programmers must simultaneously get to grips with many different aspects of programming, such as tools, concepts, and syntax. This can be overwhelming and may easily lead to confusion and frustration early on. The complexity of turning even seemingly simple ideas into working programs can be truly discouraging. This has led to extensive research on how to ease the first stages in learning to program. Programming environments designed specially for novices are commonly used to reduce this complexity in the initial stages. These provide a simplified user experience with extra support features for novices. Instead of trying to come to terms with everything at once, novices can then build their knowledge and skills gradually as they have less things to deal with at a time. As with skills of any kind, practice is fundamental in learning to program. By letting learners exercise programming skills in this simpler context, a novice-oriented environment also facilitates achieving an immediate sense of accomplishment in programming. This will allow learners to build confidence before moving on to deal with the full complexity of programming languages and tools. The first high-level research question in this thesis deals with the design of programming environments for novices.

RQ1 *How to minimize the barrier to start practicing programming?*

This question is essentially about how to enable getting immediately into exercising core programming skills and ideas without a steep initial learning curve. There are two sides to this. On the one hand, we can try to initially eliminate the use of professional programming tools or in some way facilitate their use and, on the other hand, we can try to facilitate the process of understanding the execution of programs or the process of composing programs. The first aspect deals with tailor-made novice environments or novice perspectives in professional tools that limit available functionality and, overall, aim to minimize any overhead from installing and learning to use development tools. The second aspect is about such approaches as providing educational visualizations of program execution, or simplifying the mechanics for describing programs in order to initially focus more on some specific aspect like the logic of programs instead of the fine details of syntax. There is, though, one more aspect to productive practice and that is prompt feedback. To make good progress in learning to program we need feedback on our work. However, it is not possible to provide a lot of individual human guidance on large courses. We must rely on technology to fill in and, consequently, automated feedback in programming exercises has long been an active area of research in programming education.

In this thesis, we have studied three particular approaches to lowering the barrier to start practicing programming. First, we have studied the use of web technologies in implementing programming environments for novices in order to make these more easily accessible. Second, we have studied the use of program construction exercises in teaching programming. In these exercises, programs are not written from scratch but the process is initially simplified by letting learners construct programs from a smaller set of appropriately chosen code fragments. In particular, we have worked on how to improve automated feedback in program construction exercises. Third, we have studied the use of touch devices for practicing programming in order to support mobile and ubiquitous learning. Specifically, we have worked on how to adapt program construction exercises to mobile touch devices. From a technology point of view, the underlying theme for all of these approaches is the design and implementation of environments for practicing programming.

Web-Based Programming Exercises

There is a variety of reasons why we see the web as a possible enabling technology for an improved learning experience when it comes to programming. To start with, unlike traditional native applications, web applications are not tied to a single computing environment. At its best, no installation is required besides having a browser which you generally have on any platform anyway. You can also use any computer, such as one at your school or the one at your parents' place, and the user experience should be more or less identical across platforms and computers. Furthermore, you do not need to deal with different versions of the application or updates. That is all managed in a centralized fashion by someone else. When the tool is available online, you can access it anywhere where there is an internet connection. It can automatically store your progress remotely and you could then continue anywhere on a different computer without having to worry about storing and transferring your work. Due to reasons like these, there is an ongoing general trend towards web-based cloud computing. As a logical culmination of this, the Google Chrome OS[2] operating system does not even have almost any other native applications besides the browser and web applications are used to provide most functionality.

Another issue to consider is the overall organization of learning content and tools. The use of books and other printed material is becoming less and less common and electronic learning materials are taking over. Instructors distribute their materials and assignments through web-based learning environments and learners search online for documentation and additional information. As for programming specifically, learners regularly make use of web-based systems to get automated feedback on their programming assignments. One approach is also the use of interactive web-based tutorials where theory, examples, and exercises are intermixed. All in all, a programming environment built with web technologies could presumably be seamlessly integrated into the online learning environment, with the automated feedback system, and within web-based learning content itself. Similarly, programming exercises built on top of such a web-based environment have the potential of requiring less effort to share and disseminate. In the scenario where different learning materials and tools were to be brought together in a technology sense, there would also be an additional opportunity to keep track of and study the links between those. As a data source for learning analytics this could significantly benefit our under-

[2] `http://www.google.com/chromeos`

standing of the learning process. Such data could potentially also be used in automatically adapting the learning experience, such as by recommending learning topics based on the individual progress in terms of all the different types of content. More detailed knowledge of learners' progress could also possibly be used by instructors to initiate early interventions when problems arise.

Massive Open Online Courses, or MOOCs as they are generally called, provide another perspective to this discussion of web-based learning content. MOOCs are essentially distance learning courses open to anyone with no limit to the number of participants who also often are geographically dispersed[3]. In such a learning context, web technologies have become the backbone for organizing learning activities. In fact, it is exactly the advances in web technologies and web-based learning technologies that have mostly given rise to the recent emergence of MOOCs. Similarly, a programming environment built with web technologies would be ideal for a MOOC on programming. Learners would be guaranteed to have access to a common working environment with only minimal requirements from their own computing environments and the instructor would for the most part only need to deal with this single configuration of tools in the teaching materials.

As a conclusion to this discussion, the research question is as follows.

RQ1.1 *Can we, using web technologies, lower the barrier to start practicing programming?*

In particular, we have studied the use web technologies in implementing and integrating supporting technologies, such as program visualization and automated feedback, into a single web-based programming environment. Previous work on program visualization, automated feedback, and programming environments for novices will be discussed in Sections 2.2, 2.3, and 2.4. In Section 3.1, we summarize Publication I and Publication V that deal with this question.

Automated Feedback in Program Construction Exercises

Program construction exercises scaffold the learning experience by letting learners *construct programs from short fragments of code* instead of *writing code* from scratch. Typically, such a code fragment is a single line of code or a few lines of code, that is, a small block of code. The terminology is ours and stems from the perception that, in these exercises, programs

[3] For example, Coursera offers MOOCs at `https://www.coursera.org/`.

are constructed – pieced together from the given set of building blocks instead of being conceived “out of nothing”. Exercises that fall under this definition have been called by many names in the literature, such as Parsons puzzles [105], code sorting[4], and code scrambles [114]. In the papers included in this thesis, we have mostly referred to these exercises as Parsons problems but in this text we adopt the much more descriptive term “program construction exercise” to refer to the assignments where learners are to perform program construction tasks of this kind. Where the distinction is needed, we refer to to the conventional types of programming tasks where learners write code “character-by-character” as *code writing exercises*.

In the simplest form, the learner is given a minimal set of code lines needed to implement some described functionality. In this case, the task may reduce to that of ordering those lines correctly. This in a way shifts emphasis from the details of syntax to the general logic and flow of the program because proper syntax has already been followed within each line. An apt use case for these exercises that have the single unique solution is using them to complement example programs in self-study materials. Example programs often follow after new concepts or constructs have been introduced in instructional texts on programming. However, once the basic theory has been introduced, a program construction exercise could, as well, be used to engage the learner into applying what they have just learned in actually putting together a program. There would be no actual choice in the implementation but the learner would still presumably have to think about the structure of the program more thoroughly than when glancing over an example program given to them in full working form. In the same manner, program construction exercises might be used to illustrate some clever or intuitive solutions to specific programming problems as opposed to just handing these out or expecting learners to be able to come up with everything from scratch. The eureka moment of getting the pieces together and grasping how the program works, if that happens, might just be enough to make a more permanent imprint on their memory.

Additional code fragments may also be given that make several alternative solutions possible. To make the task more demanding, the additional fragments may also be unnecessary for completing the exercise and be there just to distract the learner. A reason for doing this would be to make learners confront their missing or misconceived knowledge. For

[4] In the ViLLE learning environment, `http://ville.cs.utu.fi/`.

example, similar lines of code with correct and incorrect syntax may be provided [105]. Alternatively, we could provide extra code that may be used to solve the exercise in a manner that is consistent with some known misconception about the concepts or constructs involved.

Small variations can also be useful at the exercise level. Exercises using the same construct in various programs and contexts may help learners to more thoroughly familiarize themselves with its function. The key idea here is that seeing different cases ought to reduce the likelihood of ending up with a flawed understanding of the behavior just because that perceived model of behavior made sense in one instance. Program construction exercises are, of course, not required for this. The exercises could just be regular programming tasks but this kind of repetition would arguably be far less tedious with the build-from-blocks approach.

In this work, we have studied the use of program construction exercises where the code fragments given are Python code lines. In Python, the block structure of programs is defined using indentation instead of more conventional cues like braces or keywords signifying the beginning and end of a block. Taking advantage of this, in the exercises we have used, learners must place code lines in their proper place both in terms of the order and the indentation. This arguably is a more demanding variant of construction exercises. In experimenting with these exercises in teaching, we have found a number of issues with the automated feedback provided in previously existing implementations. From our experiences emerged the following general research question.

RQ1.2 *How to improve automated feedback in program construction exercises?*

Previous work on related approaches to and systems for simplifying the mechanics of describing programs will be discussed in Section 2.4.1. In Section 3.2.1, we discuss specific issues with automated feedback in program construction exercises and refine the research question as we summarize Publication III and Publication IV that deal with this question.

Program Construction Exercises on Touch Devices

There is one more approach we have studied in our quest for minimizing the barrier to start practicing programming. In search of making practice possible anywhere and anytime, it is illuminative to realize that, today, it is actually very common for people to carry with them a “computer” everywhere they go. Indeed, small mobile devices like phones and tablets are used to consume all kinds of interactive material on-the-go. As such, they present an attractive platform for mobile and ubiquitous learning, as well. However, the new generations of mobile phones and tablets all rely on touch screen -based input and this interaction method is very different compared with the traditional mouse and keyboard combination of personal computers. Other challenges include the hardware limitations of screen size, performance, and battery life, but also the constraints imposed by the typical use contexts which mean that the tasks must generally be quite brief and self-contained or easily resumable to permit sporadic learning sessions. The question boils down to how to effectively create or adapt content to meet the requirements and overcome the limitations of this quite a different learning space. The research question is as follows.

RQ1.3 *How to practice programming on mobile touch devices?*

A key challenge for practicing programming is the lack of a physical keyboard on touch devices. Additionally, the research question specifically deals with *mobile* touch devices meaning that our focus is on small devices, such as phones. The approaches presented are however generally applicable to tablet-sized devices, as well. Anyway, it turns out, program construction exercises are well-suited to this scenario. Previous work on other approaches to and systems for practicing programming on touch devices will be discussed in Section 2.4.3. In Section 3.2.2, we further refine the research question as we summarize Publication III and Publication VI that deal with this question.

1.5.2 Recording and Analysis of Exercise Sessions

The process of how novice programmers solve programming assignments is generally invisible to teachers and educational researchers alike. Typically we only see the end result and maybe a few snapshots along the way. An improved understanding of the working habits and of the steps taken and difficulties encountered could be of great benefit. Face-to-face observation where the programmer narrates his or her actions during the *exercise session* provides the most complete data. Recording the computer screen is easier to setup and far less intrusive. Neither approach, however, scales well because of the effort required for data collection and, especially, for the analysis of audio and video that follows. The second high-level research question deals with this issue.

RQ2 *How to, automatically and unobtrusively, record programming* *exercise sessions and what can we learn from such data?*

The method we have studied is to instrument the exercise environment to automatically record what is being done within it. This approach has the additional benefit of potentially making it possible to analyze progress in real-time in order to give immediate guidance when appropriate. From a technology point of view, the underlying theme is the design and implementation of the recording and analysis of exercise sessions. We will use the words *interaction trace* or simply *trace* to refer to the sequence of data recorded about the actions performed in an exercise environment. We have examined traces of both program construction and programming (code writing) exercise sessions which have been collected using the systems under study as we have been addressing the first high-level research question. This is where the two research themes link together. Automatically recorded traces are also a method to learn about the use of these tools. Previous work related to this question is discussed in Section 2.5. In Chapter 4, we further refine the research question as we summarize Publication II, Publication IV, and Publication V that deal with this question.

1.6 Structure of this Thesis

In the next chapter, we discuss previous work related to this thesis. In Chapters 3 and 4 we summarize the articles this thesis consists of. These two chapters deal with the two high-level research questions *RQ1* and *RQ2* separately. Chapter 3 summarizes our work on lowering the barrier to start practicing programming and Chapter 4 summarizes our work on the recording and analysis of program construction and programming exercise sessions. Finally, Chapter 5 concludes the thesis with a discussion of key results in relation to previous work and ideas for future research.

2. Related Work

2.1 Learning and Teaching Programming

There are several approaches to the teaching of programming. Typically though, the assessment artifacts and desired outcomes of the learning process are simple programs or software systems that the student creates with the support of group teaching sessions and written learning materials. Lahtinen et al. list five typical learning situations: lectures, exercise sessions in small groups, practical sessions, studying alone, and working alone on programming coursework; and six learning material types: programming course book, lecture notes/copies of lecture slides, exercise questions and answers, still pictures of programming structures, and interactive visualizations [82]. Lahtinen et al. carried out an international survey of over 500 students from 6 different universities. Students perceived practical learning situations more useful than lectures in learning to program. As for learning materials, they considered example programs to be the most useful.

In summarizing the literature related to the capabilities and difficulties of novice programmers, Robins et al. discuss that while the literature has identified such language features as, for example, loops and recursion, as especially problematic, several authors have suggested that the most important deficits relate to underlying issues in problem solving, design, and expressing a solution/design as an actual program [117]. Consistent with students' perceptions in the survey above, they go on to note the importance of practical work by stating that frequent practical programming exercises are central in addressing this issue.

2.1.1 Theoretical Foundations of Learning to Program

Fundamentally, programming is a complex mental process where we form a mental image of an algorithmic solution and formulate that into a program by following the syntax and semantics of a programming language. At heart, programming is about program comprehension [142] – understanding programs and software in general. Be it reading or writing programs, the programmer must build a *mental model*, an internal representation of the program's intent, its data and execution, through some cognitive processes. Indeed, programming is, primarily, a cognitive skill. Schulte et al. give a good critical review of the different theoretical models of program comprehension [123].

In his articles Ben-Ari [12, 13] applies constructivism to computer science education (CSE), and concludes that "Given the central place of constructivist learning theory and its influence on pedagogy, computer science educators should . . . analyze their educational proposals in terms of constructivism". Constructivist learning theory can, indeed, shed some light on the learning process. The basic tenet of the theory is that knowledge is constructed based on observations on top of what is known beforehand. In other words, learners build their understanding by combining what they observe with their pre-existing models of knowledge. This is to say, that learners do not simply absorb fully structured knowledge presented to them but that they try to merge it into their current understanding. In essence, the theory claims that "all learning involves the interpretation of phenomena, situations, and events, including classroom instruction, through the perspective of the learner's existing knowledge" [130]. If the models resulting from the learner's interpretation prove "adequate in the contexts in which they were created" [149] those models are *viable*. That is, they allow the learner to accurately and consistently explain the phenomenon under study. On the other hand, this constructivist view then suggests that, as learners each construct their own understanding, they may frequently end up with a *misconception*, with a model that is not viable but instead fails to explain some aspect of the phenomenon. Moreover, because learners build knowledge on top of models that they believe reflect a phenomenon accurately (enough) we cannot simply shove a different model in place of the existing understanding but instead "the goal of instruction should be not to exchange misconceptions for expert concepts but to provide the experiential basis for complex and gradual processes of conceptual change" [130]. Confronted by a situation where the learner's existing model fails to explain a phenomenon, this cognitive conflict may then lead to revision of that model to a viable one. Ben-Ari expresses the same constructivist idea: "Teaching how to do a task can be successful initially, but eventually this knowledge will not be sufficient. . . . The teacher must guide the student in the construction of a viable model so that new situations can be interpreted in terms of the model and correct responses formulated." [13]

Getting back to programming, what this implies is that in order to learn to program you must experience it in a way that leaves no room for misconceptions. A way to do this is repeated practice that "covers all the angles" – practice that adequately illustrates the underlying computational model in order to construct a model of knowledge that is able to make accurate predictions on the behavior of programs. On the other hand, while programmers can arm themselves with an array of pattern approaches that can be applied in many situations, ultimately, each problem will have a unique solution consisting of many such building blocks. Rote learning of bits of knowledge that can be pieced together is possible only to a limited extent. However, experts develop and view programs as being made up of instantiations of abstract programming plans, *schemas*, such as iterating through a container and counting items [117].

Essentially, the model that programming builds on is that implied by the programming language's constructs, the notional machine [40]. It is a conceptual understanding of the computer's execution model in the context of a particular programming language – how different constructs in the language affect control and data flow. Comprehension of programs is fundamentally built on this mental model, and aside from aspects like dealing with programming tools and following a systematic process, teaching programming is about facilitating the construction of a viable model of the notional machine by various means some of which have been studied in this work.

Sorva provides an in-depth review and summary of the literature on programming misconceptions, the cognitive theory of mental models, constructivist theory of knowledge and learning, and other more recent theories related to learning programming [133]. He concludes that as a whole "the literature points to notional machines as a major challenge in introductory programming" and argues that "instructors should acknowledge the notional machine as an explicit learning objective and address it in teaching".

The work in this thesis deals with a few means of facilitating this learning process. The goal of program visualization is to give an illustration of the notional machine so that learners may compare and revise their own mental models. Program visualization will be discussed in Section 2.2. Automated feedback aids learners to make progress when practicing programming on their own. Automated feedback will be discussed in Section 2.3. On the other hand, novice environments for programming make use of several different approaches. For example, they may initially simplify some aspects of programming, such as the use of programming tools or the syntax of the language, in order to reduce the cumulative cognitive load imposed by the many aspects of programming. Novice environments are discussed in Section 2.4. Finally, apart from the cognitive skill of making sense of programs, the process aspects of programming are discussed in Section 2.5.

2.1.2 Computer Science Education Research

Computer science education research is the field of research where the work in this thesis is situated. Essentially, it is the study of theories, methods, and tools for teaching and learning computer science. Fincher and Petre have identified ten broad topic areas: student understanding, animation/visualization/simulation systems, teaching methods, assessment, educational technology, the transfer of professional practice into the classroom, the incorporation of new development and new technologies into the classroom, transferring to remote teaching ("e-learning"), recruitment and retention of students, and the construction of the discipline itself ([45], pages 3–7). As a key subject in computer science, the study of learning and teaching programming is an active area of this field of research. In a review of this sub-field, Robins et al. list some common topics: program comprehension and generation, mental models, and the knowledge and skills required to program [117]. Whereas the literature survey of introductory programming by Pears et al. groups research into four categories: curricula, pedagogy, language choice, and tools for teaching [109]. Curriculum deals with what is taught, pedagogy with the manner in which teaching and learning are carried out, and language choice with the alternative programming languages. Finally, the fourth category, tools, is where most of the work in this thesis falls into, and the general research topic is supporting the acquisition of programming skills.

2.1.3 Software for Learning and Teaching Programming

Programming tools are generally developed to meet the needs of professionals and novice programmers are very different from experts. Extensive sets of features and lack of scaffolding that would allow novices to try and grasp the complex process of developing software in smaller progressive steps make these tools less suitable to beginning programmers. The literature survey of introductory programming by Pears et al. groups tools into four categories: visualization, automated assessment, programming environments, and other tools [109]. The last category includes, for example, plagiarism detection and intelligent tutoring systems. This thesis relates to research in the first three categories and each of them will be discussed in their own section. The other aspect of this work, collecting and analyzing data on the process of solving programming assignments, will be discussed in the last section of this chapter.

2.2 Program Visualization

Software is inherently invisible and the concepts in programming are abstract. As with any abstract constructs, illustrations can be used to try to convey information about them. Indeed, software is commonly visualized in an attempt to facilitate program comprehension and support software engineering activities.

Software visualization (SV) is "the visualization of artifacts related to software and its development process" [38], and is used in the presentation, navigation and analysis of software systems. This wide definition includes, but is not restricted to, the visualization of program code and data, requirements and design documentation, source code changes, bug reports, software quality and other metrics, and testing results. Specific areas of SV that have been widely applied to programming education, and are therefore most relevant to our discussion, are the areas of program and algorithm visualization. *Program visualization* (PV) refers to the visualization of the *source code*[1] and data of a program [113]. *Algorithm visualization* (AV), on the other hand, is understood to mean the visualization of algorithms and programs on a higher level of abstraction, that

[1] "Source code is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form." [19]

is, on a more conceptual level [113]. Karavirta et al. provide a taxonomy and review of algorithm animation [67]. The work in this thesis does not, however, deal with algorithm animations.

As for program visualization, state of the art visualization systems present a step by step visualization of the execution at the level of expression evaluations (e.g. Jeliot 3 [14, 99], UUhistle [136]) and possibly more abstract visualizations of data structures (e.g. jGRASP [28], Online Python Tutor [55]). Some recent systems have joined the trend of web-based applications and are using HTML5 and JavaScript to implement the visualizations (e.g. Online Python Tutor [55], JavaScript library for visualizing program execution [128]). For a comprehensive recent review of program visualization systems intended for teaching beginners about the run-time behavior of computer programs see [134].

Role of Engagement in Educational Visualization

While intuitively program and algorithm visualizations seem like powerful teaching methods this is not necessarily true. Their effectiveness has been suggested to depend on the chosen level of abstraction and how clean the visual presentation is [120]. Other research on educational AV also indicates that the degree of user interaction with the visualizations, i.e., the learner engagement, is a major factor as far as actual learning is concerned. In an extensive study on existing research on algorithm visualization, Hundhausen et al. [58] concluded that the most beneficial uses of AV are those that activate the student, for example, with questions or exercises, as opposed to plain viewing. They suggest that AV is most effective when used in a supporting role of some engaging activity. Naps et al. [101] go as far as to say that educational SV is of little value if it does not engage the students in an active learning activity. This result has encouraged researchers to create systems that integrate software visualization and automated assessment: students are given tasks related to a visualization and their answers are automatically evaluated for correctness to give them immediate feedback. This way, the systems are able to engage students more effectively. For example, recent work has explored visual program simulation as a task that combines visualizations of execution and an engaging activity [132, 135, 136, 137]. Learners are given programs whose execution they are to simulate using a graphical representation of the computational model of Python.

Discussion

This thesis draws upon research in program visualization and presents Jype, a new program visualization and programming exercise tool for Python. Compared to the current state of the art systems, such as Jeliot 3, UUhistle, and jGRASP, the visualization features are not quite as advanced. The tool provides only a line-by-line visualization of execution instead of being able to step through each step in the evaluation of expressions like in Jeliot 3 and UUhistle. The data structure visualizations in jGRASP are more sophisticated as well. Nevertheless, at the time the tool was developed, there were no comparable tools for visualizing Python execution. Today, UUhistle and Online Python Tutor are excellent alternatives for visualizing Python execution.

Furthermore, there are two overall themes in this thesis, first of which deals with lowering the barrier to start practicing programming. Visualizations of execution should help in this but also, in view of this, a key goal in this visualization tool was to make it as easily accessible as possible. This was approached by creating an integrated web-based environment for programming exercises coupled with automated assessment and the program visualization features instead of having separate tools for each task. As discussed in this section, educational visualizations may be more effective when learners are not merely viewing them but actively engaged with them. Along these lines, program visualizations might be more beneficial when closely combined with the programming tasks. Certainly, while there are some merits to the visualization features provided in Jype, in regard to related work, the primary contributions of this work lie in the unique combination of features it provides in an integrated web-based tool. The work in this thesis dealing with this tool will be summarized in Section 3.1.1.

2.3 Automated Assessment and Feedback

Typical tasks for a beginning programmer include writing, extending, or modifying a simple program or a piece of code. Going through and grading these students' submissions is a time-consuming and mostly monotonous endeavor which quickly becomes a major burden on the teaching staff with their often strict resource constraints. Of course, one could simply provide optional exercises that are voluntary to complete and are not checked by the course staff. However, the applicability of this approach is very limited. In their discussion of assessing programming assignments on large courses, Ala-Mutka and Järvinen note the tendency of students to try to minimize their workload [4], which renders optional activities in the learning process ineffective. Woit and Mason carried out a five year study comparing different assignment strategies, where two experiments contained optional assignments [152]. Along the same lines, they reported that their students mostly ignored optional tasks, which was also directly reflected in poor midterm exam results. In other words, the programming exercises must be mandatory to really make a difference, in which case we also need a way of, at least on some level, checking that the result is acceptable.

In computer assisted assessment (CAA), the evaluation process is supported with software tools that fully or partially automate the tasks involved in order to reduce effort and speed up the process. Examples vary from facilitating rapid and consistent grading with feedback authoring tools (e.g. [3]) to intelligent tutoring systems (e.g. [10, 98, 143]) that simultaneously monitor and model the progress of and guide and give intelligent feedback on the student's learning. For a quite recent review of automated assessment tools for programming exercises see [60].

An alternative approach to practicing programming concepts is to focus on analyzing and tracing programs instead of program synthesis, that is, instead of writing programs. For example, in TRAKLA2, using a drag-and-drop interface, learners manipulate visualizations of data to simulate how an algorithm works [89]. In UUhistle, learners manipulate visualizations of Python's notional machine to simulate program execution [136]. In ViLLE, learners answer questions about the execution of programs while it is being visualized [116]. Then again, Problets is a whole family of pioneering learning tools where learners must analyze and trace programs and then solve a variety of types of problems to test and demonstrate their understanding[2].

Problets is essentially a suite of tutor applications each of which provides exercises for a single topic, such as, for example, pointers or `for` loops. Additionally, instead of simply providing a platform and a set of exercises for practicing the concepts, each tutor tries to adapt to the needs of the learner by presenting new problems based on the learner's previous performance until the learner is considered to have mastered the topic [77]. Tutors have been created for a variety of mostly CS1 topics: selection

[2] `http://www.problets.org/`

statements [127], scope concepts [76], C++ pointers [78], parameter passing mechanisms [125], loops [29], classes [44], expression evaluation [79], arrays, and functions. Most of these tutors are currently available for the Java, C++, and C# languages[3]. A few are also available for Visual Basic. Each tutor has its own types of exercises and user interfaces for solving them but the tasks are generally about predicting the behavior of given code – specifying output or bugs. The tutors grade the learner's answers automatically and most of them provide as feedback a step-by-step written explanation of program execution [80]. As a somewhat unique feature, the problems presented to the learners are not a fixed set but each problem is an instance of a parameterized template generated by assigning random values to specified parts of the template [80]. Thus, there is a seemingly endless supply of different problems. The tutors are delivered via the web as separate Java applets.

Discussion

All the publications in this thesis directly or indirectly deal with automated assessment of programming exercises. We present tools for practicing programming that support automated assessment and analyze data from such tools. In these tools, the tasks are about program synthesis – on composing programs that meet some given requirements. We have studied two specific types of exercises: code writing and program construction. With regard to code writing exercises, the focus is not on advances in automated assessment but in order to try and minimize the barrier to start practicing programming, the work in this thesis deals with web-based systems that integrate editing code and automated assessment. Several related tools will be discussed further in Section 2.4.2. The work in this thesis dealing with this theme will be summarized in Section 3.1. As for program construction, this thesis work presents improvements to the automated assessment in these exercises. In section 2.4.1, we will discuss related work on program construction exercises and their automated assessment. The work in this thesis dealing with this theme will be summarized in Section 3.2.

[3] `http://www.problets.org/about/topics/index.html`

2.4 Programming Environments for Novices

Programming is a complex task that builds on several layers of abstractions of hardware and software. To facilitate software development, programmers utilize a chain of various tools. At the very least, the programming environment must provide the capability to edit, build, and execute programs. Integrated development environments (IDEs) are applications that integrate these functions and typically also provide additional features, such as organizing work and resources into projects, a visual debugger, testing and refactoring tools, syntax highlighting, code completion, integration to version control, and so on. However, when learning to program, the advantages of a general-purpose, professional IDE may well be outweighed by its complexity and the resulting steep learning curve. For this reason, many kinds of programming environments aimed at novices have been developed.

The literature survey on the teaching of introductory programming by Pears et al. groups education-oriented programming environments into two broad categories: *programming support tools* and *microworlds* [109]. Programming support tools typically limit the set of features available while also streamlining their use (e.g. [5]). They may also provide additional visualizations of code or execution (e.g. [66]) or interactive incremental execution (e.g. [5]). Editing programs and following proper syntax may, as well, be supported. BlueJ [74] is a popular education-oriented IDE for Java. Its primary features are visualization of the class structure as a static UML diagram and the ability to create objects and call methods on them via a simple GUI in order to examine their behavior. Another type of tool is the tiered language tool in which novices can use more sophisticated versions of a language as they learn more (e.g. ProfessorJ[4] [52]).

Microworld environments present programming in terms of a physical metaphor – the microworld – such as a robot moving about in an environment based on the given set of commands – the program. The world can be virtual, i.e. a graphical representation on the computer screen (e.g. Jeroo [122], JKarelRobot [21], PigWorld [84], GreenFoot [73]), or a real physical environment (see e.g. [91]). Objects in the environment then reflect the state of computation.

As another characterization of this category of tools, Kelleher and Pausch have devised a taxonomy of novice programming environments and lan-

[4] `http://www.professorj.org/`

guages [70]. At the top level, the systems are classified into two types by their primary goal: *empowering systems* and *teaching systems*. Empowering systems support programming to facilitate accomplishing tasks to meet some need of the user, i.e. they enable end-user programming, or exploring something in an entirely different domain, like e.g. cognitive modeling in a psychology class (e.g. [26]). The ability to program is simply a tool – not the end goal. Most importantly, the designers of these systems are not concerned with whether the users can transfer this practice to a broader programming context. Thus, they may invent a special-purpose language that has a very different vocabulary and grammar compared with any widely used general-purpose languages or forget code altogether and let programming be performed e.g. by demonstrating rules and actions with a graphical interface. Teaching systems, on the other hand, provide exposure to some of the fundamental aspects of the programming process in an attempt to facilitate learning generalizable programming knowledge and skills – to teach programming for its own sake. The goal all along is that eventually, as their skills improve, users move on to a general-purpose, professional environment. An important point made by Kelleher and Pausch is that these systems must strike a balance between two conflicting goals: making it easier for beginners to get started in programming, and still building the background that eventually allows them to successfully transition from the teaching system to the professional tools. This thesis deals solely with teaching systems and empowering systems are outside the scope of this work.

Kelleher and Pausch further divide teaching systems into two categories: mechanics of programming and learning support [70]. The latter category deals with systems that try to improve learning in other ways, such as by providing a motivating context or facilitating learning in groups and from peers. For example, games have been used in trying to provide a more engaging context for programming (e.g. [85]). The work in this thesis falls in the first category and the latter will not be discussed any further. The systems in the first category focus on simplifying the process of writing programs and of understanding the execution of programs. While this is not discussed in the taxonomy, any programming environments with support for visualizing program execution fall into this category. This topic was discussed in Section 2.2. Additionally, Kelleher and Pausch group microworlds, that were discussed above, here.

To sum it up, in terms of the first categorization, the work in this thesis deals with programming support tools, and regarding the taxonomy, we have specifically explored lowering the barrier to start practicing programming by simplifying the mechanics of programming. We have approached this in three different ways. First, we have experimented with exercises where the mechanics of describing programs are simplified. Related work in this area is discussed in the next section. Second, we have experimented with simplifying the programming environment via the use of web technologies and integration of limited key functionality, such as assignment delivery, editing and executing code, and automated feedback. Related work in this area will be discussed in the section following the next one. Finally, aside from the categorizations, we have explored a novel approach of enabling the use of mobile touch devices, such as phones and tablets, in learning to program. In the last section, we discuss related work in the emerging field of practicing programming on mobile devices.

2.4.1 Simplifying the Mechanics of Programming

Typically, programs are created by typing textual instructions. Novices often have difficulties in following the structure and syntax of a programming language in formulating their intentions into computer instructions. There are two steps that can be made easier for novices: the language can be simplified or the input method can be made simpler. Programming language design involves various different competing constraints in addition to readability and intuitiveness, such as versatility and the ease and performance of implementations. This is why professional languages may be unnecessarily complex for learning the basics of programming. Approaches to simplifying the language range from inventing an entirely new language to hiding functionality that is then revealed progressively. As for typing code, there is a variety of alternative methods.

In ProPAT, learners complete C programming exercises with the help of a library of programming patterns they can select to add to their programs [31]. As additional guidance, the tool will only allow patterns to be inserted in a way that the C syntax rules are followed. The patterns are textbook solutions to common problems, such as a counting loop, described in a way to ease reuse. The theory behind is that experienced programmers similarly apply knowledge of previous similar solutions to solve the problem at hand but novices lack such experiential basis. The system is implemented as an Eclipse plug-in.

PatternCoder[5] is another tool that supports writing code with a library of patterns [106]. However, these are higher-level design patterns and the aim is to aid learners in understanding class associations and in transitioning to a class-level design of code. The wizard-based interface allows learners to select appropriate design patterns for their problem and generate the respective Java code that they can then explore and use as starting point. The system is implemented as an extension to BlueJ[6]. Patterns+UML is a very similar wizard-like standalone tool aimed at helping novice programmers learning object-oriented programming to implement design patterns in Java [32].

Green[7] is another tool aimed at novices that can be used to generate code from UML class diagrams [7]. It is a UML editor that allows both generation of code from a diagram and creating a diagram out of code. It is implemented as a plug-in to Eclipse. The tool has been used in introductory programming classes to support an instructional approach where strong focus is put on object-orientation, UML design, and design patterns [8]. Previously, the authors had also developed a similar tool called QuickUML for their needs [9]. There are, of course, many professional tools for generating code from UML diagrams, such as, for example, eUML2[8].

BACCII and BACCII++ represent early examples of another family of programming tools where programs are created using an iconic programming language [22, 23, 24]. Programming constructs, such as loops and conditional branching, and data are represented using graphical icons that are connected to each other to describe the control flow. The user can then generate the actual corresponding code for several programming languages. The tool was implemented for Windows. The authors report using BACCII/BACCII++ on introductory PASCAL/C++ programming courses where students performed better when the tool was available for them [22, 24]. More recent similar systems are, for example, SFC [150] and RAPTOR [25].

In Alice 3[9], programs are built by drag-and-dropping graphic tiles whose instructions are similar to those found in programming languages[10] [30].

[5] `http://www.patterncoder.org/`

[6] `http://www.bluej.org/`

[7] `http://green.sourceforge.net/`

[8] `http://www.soyatec.com/euml2/`

[9] `http://www.alice.org/`

[10] Originally the language was built on top of Python but this was changed in later versions.

Programs are used to create 3D animations and games. Learners can then reflect on their program in terms of the visual objects. Alice programs can be exported to Java code. There are, as well, several similar novice environments based on a jigsaw metaphor where programs are built from blocks that have different appearances based on their type and can be connected together in limited ways as illustrated by their forms which fit together only in certain ways. Typically, the blocks, additionally, have within them editable parts like, for example, a number that can be incremented or decremented. BLOX [50] is an early example and Scratch[11] [90], Snap![12] (formerly BYOB, Build Your Own Blocks), Blockly[13], App Inventor[14], StarLogo TNG[15], and Turtle/PictureBlocks [147] are some more recent systems. The environments are designed for exploring programming and computational thinking by making interactive visual content like games, animations, and simulations. Typically, they are also primarily geared towards children and pre-CS1, CS0-level studies. These types of systems fall into a category of tools called visual programming environments. Some of the rules and formalism of the programming language that must conventionally be followed in the textual notation are replaced with graphical representations like how the jigsaw metaphor only allows blocks to be combined in certain ways. Consequently, it is generally impossible to make syntax errors – the environment will only allow executable programs to be built.

Karel Universe is another system where you can actually "write" Java code by drag-and-dropping[16] [17]. This is an editor integrated to the Karel J Robot [18] microworld environment. The editor lets programs be built from code fragments and they can then be executed in the Karel J Robot environment. The system also only allows syntactically correct modifications.

JPie falls somewhere between these approaches [51]. The actual structure of Java programming language is used but it is manipulated using manipulation of graphical representations of the programming abstractions. The jigsaw metaphor is also used.

[11] http://scratch.mit.edu/

[12] http://snap.berkeley.edu/

[13] https://code.google.com/p/blockly/

[14] http://appinventor.mit.edu/

[15] http://education.mit.edu/projects/starlogo-tng

[16] http://www.csis.pace.edu/~bergin/KarelJava2ed/kareluniverse/index.html

Program Construction

In Section 1.5 we defined what we mean by program construction exercises and also discussed what they are for. Essentially, learners construct programs from blocks of code instead of writing from scratch. In this section, we discuss work on assignments that fall into this category.

Originally, Parsons and Haden conceived, or at least published the concept of programming tasks where you are given a list of code fragments that must be reordered to build a program [105]. They then became called Parson's programming puzzles. The code fragments were a single or a few lines of code. The exercises presented a sequence of randomly ordered code blocks that could be drag-and-dropped to a another sequence of slots to build the program. There could be more alternative code fragments than how many slots there were, that is, there could be extra code. The number of slots also revealed how many blocks a correct solution contains. The implementation was based on a generic web framework for multiple choice questions called HotPotatoes and the answer was deemed correct if each slot then contained the expected code. A particular weakness in using such a framework is that the interface for reordering code fragments is quite clumsy as you cannot just drop code in between but have to move other fragments one by one to different slots first. The original idea with Parsons puzzles was to support computer-assisted rote learning of syntax.

CORT, COde Restructuring tool, is a standalone windows application that is used to provide tasks similar to Parson's puzzles [47]. However, in these tasks there is a focus on providing a fixed context around the area where the learner adds code. In other words, there is some "static" surrounding code. The author calls this the part-complete solution method and it is meant to reduce cognitive load compared to a full-fledged programming task. Lines of code can then be added to the assigned area from a given set of options where there can be extra code like in the original Parsons puzzles. Using CORT a learner gets feedback by copy-and-pasting their solution to an interpreter to see what happens.

In more recent work, web-based ViLLE (the visual learning tool, [81, 116]) learning system for programming has in later versions added support for similar code sorting exercises[17]. However, in these exercises the task really is to sort a set of code lines "in-place" and there can be no extra code. These exercises are implemented in JavaScript and provide an easy-to-

[17] The online version of ViLLE includes code sorting exercise while they are unpublished: `http://ville.cs.utu.fi/`.

use drag-and-drop interface. Furthermore, learners can get automated feedback on their attempts to solve an exercise. Feedback is a score which we assume is based on running the code remotely and comparing its output to the expected output. Code that fails to execute is given a zero score. As a unique feature, if the program can be executed, learners may also visualize its execution line-by-line in order to debug their solution.

In other very recent work, similar sorting exercises that are being called code scrambles have been included in an open web site for learning Python [114]. As in ViLLE, the interface is implemented in JavaScript and code lines are only sorted – there can be no extra lines. The learner may also request automated feedback and this is based on giving the results of executing tests on the program on a remote server. As possible future work, the authors speculate that these code scramble exercises might also work well or even better on a touch screen -based device.

Finally, js-parsons is a quite recent open source[18] library and web-based environment for similar Python exercises [61]. This was also implemented in JavaScript and provides an easy-to-use drag-and-drop interface. In Python, indentation is significant and is used to define blocks. Making use of this, a unique feature in js-parsons is that the learner must arrange the code fragments in terms of two dimensions: the order and the indentation. This makes the task more complex compared to the other systems. The system also supports extra lines and can provide automated feedback. In the original implementation, feedback was based on comparing the fragments first to an expected ordering and then to the expected indentation of those fragments. In other words, the constructed program was examined line-by-line starting from the top and the first line that was found to be incorrectly placed became highlighted, and if this check passed the proper indentation was checked in the same manner starting from the top line-by-line. Additionally, js-parsons can be used to log how the exercise is solved and store this to a remote server for later analysis.

Apart from all these software tools, similar assignments have also been used in pen-and-paper exams [34, 86].

Discussion

In this section, we have discussed several different approaches to simplifying the mechanics of programming: tools based on giving a library of patterns to use in building programs, tools built around UML, and

[18] `https://github.com/vkaravir/js-parsons`

many different types of tools that give program code an iconic or a diagrammatic visual form in addition or instead of using text. Finally, we discussed program construction exercises. While especially many of the environments based on the jigsaw metaphor bear a resemblance to the program construction exercises and have had influence on those, a specific difference to many of the other systems is that, as we see it, these are not visual programming environments. Programs are still just textual code even though they are built from fragments of those. There is no actual visual rule of syntax. What you learn about how to create programs is still, in theory, directly transferable to actual writing of code despite the scaffolding.

Overall, with regard to the different environments, js-parsons stands out as the most versatile tool for program construction exercises. It is open source, can be easily integrated into web-based learning environments because it is a web-based self-contained client-side widget, supports extra lines, provides automated feedback, and a clean user interface. Most importantly, however, it supports a unique type of exercises where Python code fragments must additionally be properly indented and this makes one think about the design of the program in a different way than if one was to only select and sort code. In the work in this thesis, we have experimented with the use of exercises based on js-parsons on programming courses. We have also developed the tool further in several ways that will be summarized in Section 3.2.

2.4.2 Web-Based Programming Environments

An ongoing trend in application development all over has been a transition from desktop applications to web-based rich internet applications. Some applications have already, for the most part, moved to “the cloud”, such as email and instant messaging. The web-based approach has several possible advantages when it comes to programming environments, as well. Compared with the “traditional” approach a web-based programming environment would presumably provide everyone with a centralized, identical environment with no setup and instant updates, nor would it be tied to a single computer or location. The potentially lower barrier to beginning programming could be especially beneficial for a novice. On the other hand, learning environments and automated assessment systems today all have web interfaces. A web-based movice programming environment could be seamlessly integrated in terms of the user experience but also with regard to recording and analyzing learning efforts in order to provide better feedback or adapt learning content based on knowledge modeling. Consequently, various types of web-based environments that provide some support of editing solutions online and typically also integrate with automated assessment have been designed. In most of these, the aim has been to create a practice environment that allows getting a quick start and immediate sense of progress through lots of practice without first dealing with the hassle of installing and learning to use professional tools.

Javala was[19] an open web environment for learning Java programming [83] whose unique feature is its game-like scoring of users. It provides tutorials with embedded "fill in method body" -type programming exercises where the code can be written directly in the browser in a simple text box. Learners' code is assessed on a remote server by comparing its output to the expected behavior. The learner receives Javala points for each correct solution. There is a global top 100 scores list of all the users of the system and the learners are also assigned a rank, such as "Java Tourist" or "Java King", based on their performance. The author reports that the users spent very long continuous periods in the system and attributes this to the addictive nature of the game-like features. Additionally, the author discusses the submissions to a few exercises and presents visualizations of a few learners' submission activities in the system. He notes that some students seemed to get stuck and submit a seemingly simple exercise many times, and suggests that in these cases the system should give some hints on how to solve the exercise instead of simply showing the error. He also suggests that a further thorough analysis and categorization of the learners' habits could help in improving the environment.

CodingBat[20] (previously named JavaBat) is a web site that provides a large set of small programming problems for Java and Python on common CS1 topics, such as loops, branching, strings, and so on. The idea is to have very little context but focus on simply practicing the mechanics of programming [104]. In each exercise, the learner is required to write a function that implements some described functionality. Solutions can be written in a simple text box directly in the browser and may then be submitted for evaluation. The code is tested on a remote server by calling the function with different values and comparing the return values to the

[19] The author could not find the system online anymore. The system was only available in Finnish.

[20] `http://codingbat.com/`

expected behavior. Results of the tests are then shown on the web page. CodingBat also implements a gamification feature where you earn a so called badge for solving many problems[21].

The ViLLE environment mentioned earlier for having code sorting exercises, is primarily a program visualization tool whose main features are programming language independency and the ability to create animations that include multiple choice questions at specific steps of the animation. However, there are also some code writing tasks that are assessed by running tests on a remote server.

CodeWrite [36] is another web-based environment for practicing Java programming. The exercises are similar to Javala and CodingBat in that learners are to complete the body for a single method. The unique feature of CodeWrite is that the learners themselves author the exercises and test cases (input/output pairs) that will be used in providing automated feedback. Additionally, learners must provide a working solution that satisfies the tests they have defined. Learners can then choose exercises that others have designed, edit code directly in the browser in a simple text box, and submit when they are ready to receive feedback. Their code is tested on a remote server, and feedback is shown on the web page. Furthermore, after completing an exercise, other successful submissions are shown to the learner thus enabling them to compare different styles of solutions. Learners can, as well, comment on or rate the quality of an exercise or endorse solutions. The authors report successfully using the system on a CS1 course where they found that the student-generated exercises and solutions exhibited good coverage of the language features on the course. CodeWrite has also been used successfully in a cross-institutional setting where students from different universities designed exercises for each other [33].

Another recent web-based environment for learning to program is Computer Science Circles[22] [114]. Similar to Javala it has tutorials with embedded exercises but the language taught is Python. The aim has been to create a complete interactive introductory programming course using web technologies. Similar to CloudCoder discussed below, for programming exercises CS Circles provides a more capable editor[23], instead of simple text box. Additionally, CS Circles uses Online Python Tutor[24] [55] for

[21] `http://codingbat.com/doc/practice/code-badges.html`

[22] `http://cscircles.cemc.uwaterloo.ca/`

[23] CodeMirror, `http://codemirror.net/`

[24] `http://www.pythontutor.com/`

letting learners visualize the execution of their programs. Code submitted in programming exercises is executed on a remote server and graded based on generated output, values of variables, or the return values of function calls. Learners can additionally open code in a separate console window for trying out arbitrary code instead of only running the tests of the current exercise. However, the console does not support interactive execution with input and output. Finally, a rather unique feature in the environment is its help functionality. Next to the code editor is a help button which the learners can use to ask for help if they get stuck. The help request will include the current code and also provide a link for accessing the complete submission history, the progress summary of the learner, and a list of messages about the same exercise that allows the instructor to copy and paste frequent guidance. A teacher can register as a guru for his or her students to get the help requests from them. The authors have found this an efficient system for providing personal feedback when the automated feedback does not cut it for the learner. At the time of publication, the authors report almost a thousand daily visits and 7000 daily code submissions.

Runestone Interactive[25] is a similar project aimed at developing interactive web-based learning material [97]. Two "books" have been produced: How to Think Like a Computer Scientist Learning with Python: Interactive Edition 2.0 and Welcome to Problem Solving with Algorithms and Data Structures. Both deal with Python and in addition to text include embedded video clips and interactive content. Like CS Circles, there are programming exercises for which code can be edited using the CodeMirror editor. What is different with the exercises on these sites, is that the Python code is executed on the client-side in the browser using Skulpt[26], a JavaScript implementation of core Python. This means that it also works offline. Feedback is in the form of passed and failed tests as typical in comparable systems. Also, similar to CS Circles, there are visualizations of execution built using the Online Python Tutor [55]. The authors also report positive experiences on using the e-book for a semester in CS1.

CloudCoder[27] is a very recent web-based programming exercise system for Java, C, C++, Python, and Ruby [103]. Apart from many of the other systems, learners write code directly in the browser using an open source

[25] `http://runestoneinteractive.org/`

[26] `http://www.skulpt.org/`

[27] `http://cloudcoder.org/`

web-based programmer's editor[28] instead of a mere text box. This editor, among other typical things, supports syntax highlighting. The system is designed for small programming exercises and learners can submit their solution to get automated feedback. Two types of exercises are possible. In function-based exercises, where the learner is to write a single function or method, the feedback is based on calling the function/method with different parameter values and comparing the return values to the expected results in much the same way as in CodingBat. In whole-program exercises where the learner writes a complete program, the task is built around receiving input and printing some output. Feedback is then based on comparing this to the expected output using regular expressions. The system is implemented using two servers: one for the web application and a database, and one for executing learners' submissions for feedback. Python and Ruby are executed using their Java implementations Jython[29] and JRuby[30]. CloudCoder is open source and includes an open web repository for sharing programming exercises. Additionally, CloudCoder logs all of the students edits and submissions for use in educational research. Papancea et al. also report having piloted the system on a few programming courses on Java, C, or C++.

Another example is the learnpython site[31] that provides interactive tutorials about Python. Similar sites exist for Java, C, JavaScript, PHP, shell programming, and C, as well[32]. The tutorials each include an exercise where you are to write code that implements some described function. Included is code that runs test on your code and can be used to evaluate whether you have succeeded. You can also view the solution. Code is edited using the CodeMirror editor like in many other similar system. Executing code is carried out on a remote server using commercial software called the Sphere Engine[33]. There is also another site that provides programming tasks using this same software. The Sphere Online Judge[34] provides programming contest -type problems to which you can submit solutions and their correctness will be assessed by executing your code on a remote server. There are other sites for these types of problems that are skewed towards

[28] Ace, `http://ace.c9.io/`

[29] `http://www.jython.org/`

[30] `http://jruby.org/`

[31] `http://www.learnpython.org/`

[32] Links to these can be found on the site for Python.

[33] `http://sphere-engine.com/`

[34] `http://www.spoj.com/`

more competent programmers than novices, for example, topcoder[35].

There are also a number of other environments run by companies and organizations for practicing programming on the web. CodeLab[36] provides web-based programming exercises with automated assessment. CodeAcademy[37] provides web-based programming classes in many different languages and awards badges and points for completing exercises that come with automated feedback, as well. Khan Academy provides programming lessons based on JavaScript[38]. The exercises use the ProcessingJS[39] visualization library and focus on interactivity and graphics.

Furthermore, there is a constantly increasing array of MOOC providers that have offered and probably will in future too offer courses on programming topics with varying support for web-based exercises. Udacity offers MOOC-type courses on introductory programming[40] as does Coursera[41]. Udacity has an automated assessment system and so does Coursera, on some courses at least. Coursera has used the CodeSkulptor[42] programming environment on an introductory Python course. This tool is based on the CodeMirror and Skulpt libraries for the editor and the execution, respectively, similar to other work discussed. FutureLearn[43], a UK-led MOOC platform, has offered a programming course built around developing games for mobile devices[44] and will also be offering a programming-related course labeled "creative coding" in the future[45]. Miriada X offers MOOCs in spanish and portuguese and, for example, one on programming for science and engineering[46]. The French government established France Universite Numerique[47] provides, for example, a MOOC in French on iPhone

[35] http://www.topcoder.com/

[36] http://turingscraft.com/

[37] http://www.codecademy.com/

[38] https://www.khanacademy.org/cs

[39] http://processingjs.org/

[40] https://www.udacity.com/course/cs101

[41] https://www.coursera.org/

[42] http://www.codeskulptor.org/

[43] https://www.futurelearn.com/

[44] https://www.futurelearn.com/courses/begin-programming

[45] https://www.futurelearn.com/courses/creative-coding

[46] https://www.miriadax.net/web/introduccion-programacion-ciencias-ingenieria-2edicion

[47] https://www.france-universite-numerique-mooc.fr

programming[48]. edX[49], Iversity[50], OpenLearning[51], and Udemy[52] offer MOOCs on programming topics as well. Overall, MOOCs are currently a hot topic and new services are being introduced all the time. Some of the course offerings include proprietary solutions for automated assessment as well. Some courses in these services are self-paced and you can start one anytime. Others follow a schedule in a similar manner to traditional classes. Often the latter is what is actually referred to as a "true" MOOC.

Discussion

In this section, we have discussed many web-based environments that combine editing code and automated assessment. They all have their strengths and weaknesses, and their unique features. Javala and CodingBat provide some gamification features. CodeWrite crowdsources creating assignments to the learners themselves and provides some peer review features. ViLLE, CS Circles, and Runestone Interactive provide visualizations of execution. Javala, CS Circles, and Runestone Interactive integrate the exercises tightly into the learning content within tutorials or a book. CloudCoder can log students' interaction in the environment for later analysis.

A common theme is that the earlier systems like Javala and CodingBat offer only rudimentary support for editing code in the form of a simple text box, while the more recent tools provide more capable code editors based on modern JavaScript libraries, such as Ace and CodeMirror as mentioned above. The programming environments in the browser are thus edging closer towards the desktop experience.

Another important aspect to consider in these web environments is how programs are executed. Traditionally, learners' code is executed on a remote server. Sandboxing the execution so that it will not interfere with the assessment system or other learners' submissions is a particular challenge here. As browsers have become more capable, for example, because of improvements in the performance of JavaScript execution, client-side alternatives have also become feasible – Runestone Interactive executes Python code directly in the browser environment using the Skulpt library that implements the core of the language. Overall, two types of assessment approaches are typical. Most systems assess correctness based

[48] https://www.france-universite-numerique-mooc.fr/courses/UPMC/18001/Trimestre_2_2014/about

[49] https://www.edx.org/

[50] https://iversity.org/

[51] https://www.openlearning.com/

[52] https://www.udemy.com/

on the values received from function calls in test runs. As the second primary approach, CloudCoder, for example, also supports evaluating correctness based on the output produced by the learner's program.

In this thesis, we present two new programming environments that run in the browser and integrate editing code and automated assessment. The work in this thesis dealing with this theme will be summarized in Section 3.1. Their relation to previous work will be discussed in Chapter 5.

2.4.3 Programming on Touch Devices

Wide adoption of mobile devices, such as phones and tablets, has also created an interesting opportunity for learning. Lots of tasks like reading email and documents are already being performed on-the-go when sitting on the train or waiting at the bus stop. Similarly, there is great potential for supporting learning on these devices but there are, of course, many challenges too. A recent meta-analysis of studies on mobile learning gives a general synthesis of the research trends [153]. Mobile phones and PDAs are most widely being used to support mobile learning, and effectiveness and system design have most commonly been the focus of previous studies.

With regard to learning programming, early work experimented with simply having students view videos of algorithm animations on a mobile device (Apple iPod) [59]. In more recent work, a mobile application for Android devices called mJeliot has been used to better engage students in lectures [108]. During lectures the Jeliot 3 program visualization system for Java is used to execute programs and together with mJeliot can then also be used to ask questions about the execution. Students can be asked to attempt predicting parameter binding and return values of methods using their own hand-held devices. The Jeliot tool then also reports summary statistics on the correctness of students' answers. Another tool that supports similar questions on program execution is the commercial Quiz&Learn Python[53] application for iOS devices. The user is presented with timed increasingly difficult multiple choice questions related to the behavior of short programs and a score is rewarded based on how long it took to answer. Additionally, users can view line-by-line visualizations of the execution to inspect their behavior. Java Quiz[54] is a similar quiz application for Java on iOS. Questions deal with both program behavior and specific constructs of the language. There also was available, for a

[53] https://itunes.apple.com/app/quiz-learn-python/id501410339

[54] https://itunes.apple.com/app/java-quiz/id464249097

while, a Scratch [94] Viewer application for iOS but it was removed[55]. This could be used to view animations built with the Scratch environment.

Beyond viewing learning content and answering simple questions, the key challenge for actually practicing creating programs is how to edit code on a modern mobile device that is based on a touch screen. Current approaches range from custom soft keyboards with programming-oriented customizations to menu-driven programming using a specially designed language.

Aside from soft keyboards on small screen devices being cumbersome to start with, a particular difficulty in such a typing-intensive activity as programming are all the special characters used that are harder to access using a general soft keyboard. For this reason, there are many applications for a more programmer-friendly keyboard, for example, Hacker's Keyboard on Android[56], that try to facilitate entering these. Predictive typing [88] and auto-complete can also be used to facilitate typing.

Codea[57] is a commercial iOS application for creating visual interactive content, such as games and simulations, that provides some enhancements for editing code on touch devices. For one, it is built around the Lua programming language whose syntax is supposed to be easier to type. Additionally, some actions like assigning colors or images are supported with special editors using touch interaction. However, the tool is aimed at tablet devices and the backbone for writing code is the soft keyboard, which is arguably less cumbersome in this size class. There are also some animation and microworld environments aimed at children on iOS-based tablets where programming is performed using a drag-and-drop block language (e.g. Hopscotch[58], Daisy the Dinosaur[59]). Recent work has also explored the use of touch screen -based interaction methods instead of keyboards on tablets [56, 93, 115].

TouchDevelop, a Windows Phone application, provides a programming environment built around a menu-driven language designed for just this purpose [146]. More recently, TouchDevelop has also been implemented as an HTML5 application which is now available on all the major platforms[60].

[55] https://mobilewikiserver.com/ScratchHopedFor.html, https://mobilewikiserver.com/Scratch.html

[56] https://play.google.com/store/apps/details?id=org.pocketworkstation.pckeyboard

[57] http://twolivesleft.com/Codea/

[58] https://www.gethopscotch.com/

[59] https://itunes.apple.com/us/app/daisy-the-dinosaur/id490514278

[60] https://www.touchdevelop.com/

Finally, there is one game-like application on iOS tablets that provides exercises related to programming where you actually build programs. In SingPath[61], you have assignments where you choose lines from a given set of code lines to build a described program. These are effectively program construction exercises as described in Section 1.5.

Discussion
Overall, there is not a lot of previous work on how to practice programming on modern mobile devices. There are the few systems that have questions on program behavior: mJeliot, Quiz&Learn Python, and Java Quiz. Then there are a few programming environments aimed at tablet-sized devices. Additionally, some of the web-based exercise environments discussed earlier in this chapter do run on a mobile browser, such as Runestone Interactive. However, having not been adapted to small screen devices and relying on text input by typing, they are at best barely usable on mobile devices and, in practice, not at all. Using soft keyboards on touch devices is much less natural than typing on a real keyboard and this is the key challenge in writing and editing code on small mobile devices. TouchDevelop stands out with its menu-driven programming language and environment aimed at exactly this context. In this thesis, we present an environment for practicing Python programming on mobile touch devices. It provides program construction exercises with automated feedback. The implementation is built on js-parsons. The SingPath application discussed above provides similar exercises on a tablet. However, in SingPath, the lines used to build programs are already properly indented which makes the task much more straightforward. The work in this thesis dealing with this theme will be summarized in Section 3.2.2.

2.5 Programming as a Process

The art of programming is not just about the end result and there are many steps to finally coming up with a functional solution to a programming problem. In the description of programming by du Boulay, the pragmatics of programming – the skills of planning, developing, testing, debugging, and so on – is an area with a strong process aspect to it [39]. Indeed, the goals of an introductory programming course should go beyond knowledge about concepts, tools, and techniques – systematic processes for developing

[61] https://itunes.apple.com/us/app/singpath-mobile/id567470737

programs should be taught as well. This has been identified early on in the literature: “Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!” [54]. Along the same lines, drawing from mathematics teaching, in an article from 1986, Soloway explains: “What has been taught in the past is by and large not what an expert actually knows. For example, geometry students typically understand each step in a proof, as the teacher puts it on the board, line by line. However, when attempting to do a proof for homework, students often have no idea where to begin. Why? Mathematicians do not develop proofs in such an orderly, linear fashion. Rather, developing a proof is a nonlinear, search process. Unfortunately, students are not told explicitly about the nonlinear nature of proof development: They see their teacher develop a proof line by line, and not surprisingly, they think they should be able to do the same. Today, teaching topics such as geometric proofs is being revised to include explicit instruction as to the heuristics that guide proof development.” [131] Soloway was talking about the need for explicitly teaching goals and plans, that is, stereotypical canned solutions to common programming problems that experts possess, and how to break down a problem statement into various such goals with known programming plans and how to then compose a solution from these. However, the same applies to the process aspects of programming. To once more drive the point through, Gantenbein, in 1989, called out for demystifying the programming process: “Anyone with a reasonable intelligence and some grasp of basic logical and mathematical concepts can learn to program; what is required is a way to demystify the programming process and help students to understand it, analyse their work, and most importantly gain the confidence in themselves that will allow them to learn the skills they need to become proficient.” [46]

2.5.1 Learning and Teaching the Process

In their review of learning and teaching programming, Robins et al. summarize that, since programming ability must rest on a foundation of programming knowledge, typical introductory programming textbooks focus on presenting such knowledge related to a particular programming language [117]. Similarly, introductory programming courses tend to be "knowledge-driven" and the majority of the studies of programming have likewise focused on the content and structure of programming knowledge. Indeed, as Bennedsen and Caspersen also argue, the programming process is generally not addressed well in programming education [15]. Traditional static teaching materials, such as textbooks and lecture notes, are insufficient and ill-suited for the purpose of exposing dynamic processes. Moreover, Bennedsen and Caspersen note the danger of students getting the false impression that there is some linear and direct "royal road" from the problem to a clean and simple solution, if they are only shown the end product, a finished ideal solution, when, in fact, the process would have included several incremental steps of analyzing the problem and designing, prototyping, testing, debugging, and refining the programming solution. Then as students are not able to develop in such a straightforward manner, they get discouraged and may lose self-confidence and motivation to continue.

Consequently, Robins et al. suggest that the aspects of programming that are not visible in the end product should be made explicit by discussing them as a program is being developed, e.g. live on lectures [117]. Indeed, live programming performed by the instructor on lectures or in lab sessions using a projector is one possible method for making the tacit aspects of the process visible. Paxton used this technique throughout a semester to incrementally develop a single larger Java project [107]. This course was aimed at students who already knew the basics of programming but not Java. After the course, students were surveyed about this method and Paxton reports answers that highlight how it allowed students to see the complete design process, debugging process, and other process aspects. Gaspar and Langevin have also advocated the use of instructor-led live programming as a teaching method [48]. They emphasized its usefulness in showing how to develop a solution from scratch and in reminding students that correct programs are rarely written in one shot. Indeed, this was done in response to having seen students follow a process they call random programming, where students attempt to piece together programs using code borrowed from textbooks or online sources without properly understanding any of the code. In a recent experiment, live programming as a teaching method on lectures was compared to showing static code examples and it was found to be at least as good an approach [121]. Additionally, there was a statistically significant difference of the treatment group performing better on the final project than the control group. This suggests that live programming may in fact prepare students better for a larger programming endeavor as could be expected based on this discussion of the importance of exposing the process aspects of programming. Other work has explored the use of mobile devices to better engage students in the process of programming live in a lecture [108].

Class time is, of course, always limited and this may be an issue with doing a lot of live programming. *Screencasts*, narrated or talk-aloud recordings of the computer screen while performing a task, have been used as an alternative method. They have the advantage of being reusable, at least to some extent, across course iterations or courses and of allowing students to study the material at their own pace and to review (parts of) it if need be. Bennedsen and Caspersen have used such videos, what they call process recordings, in their teaching [15], Gaspar and Langevin used screencasts in addition to live programming [48], and we have employed screencasts to teach the basic use of an IDE and its debugger at our institution, as well.

In their conclusions on the review of learning and teaching programming, Robins et al. suggest that an even more important distinction than the one between novice and expert programmers is that of effective and ineffective novices [117]. By this they refer to what makes some students learn with much less effort than others. They further propose that teaching should focus more on turning ineffective novices into effective ones. Moreover, they suggest that while many factors, such as motivation, confidence, knowledge, and so on, are obviously involved, the most significant differences relate to strategies rather than knowledge. By strategies they refer to how knowledge is accessed and applied during the development process. They conclude that a deeper understanding of these both kinds of novices is required.

Discussion

Programming is a dynamic process and several authors have suggested that more attention should be paid to the learning and teaching of this aspect. Live programming on lectures and screencasts have been used to demostrate experts' processes to students. However, this is just one side of the coin. Students may still develop ineffective strategies and we must strive to gain a better understanding of how students are currently working and of the problems they are facing. The difficulties in the students' processes have to be identified in order to aid learning and teaching effectively. Unfortunately, a student's development process is generally as inaccessible to the teachers as experts' processes are to the students. Where students should have material available allowing them to learn about processes, teachers should have information available about the current processes of students. Ideally, guidance could then be even provided individually when a weakness or flaw has been recognized. This is one of the primary motivations for the work in this thesis where we have experimented with recording and analyzing students' programming sessions. Related work in this area will be discussed in the next section.

2.5.2 Studying and Guiding the Process

Akin to the live programming performed by lecturers, Gaspar and Langevin have used what they call student-led live coding to expose and correct students' processes [48]. Here, instead of the instructor, students perform the live programming in the lecture. As for research methods, one approach to collecting data about the development process is to use protocol analysis in conjunction with a think-aloud protocol where the programmer is advised to verbally express their thoughts and actions while performing a task. This can then be observed directly or recorded on video (e.g. [53]). This approach results in a very rich set of data but is very time- and labor-intensive both in terms of data collection and analysis. Another approach is to only record the computer screen either using a camera or simply a screen recording software (e.g. [57]). This data requires less effort to collect but is more difficult to analyze without the verbal trace relating the programmer's actions to his or her goals and intentions. While these two approaches provide the most in-depth data, they are highly obtrusive to the subjects being studied, are difficult to generalize to larger samples due to the considerable effort involved, and do not easily support automated analysis because of the nature of the data as audio and images. Alternative sources for data include automated assessment and submission systems and version control systems that by their nature record a coarse-grained history of the development process. A final method is to use development tools and record the interaction with any software the programmer uses, such as the compiler or the programming environment. There is a rapidly growing body of research on this topic which has also been studied in this thesis. Many of the studies mentioned here are also discussed in more detail in Publication V.

Early Work

In the seminal work by Spohrer and Soloway as far back as in the 80s, an instrumented operating system was used to record each syntactically correct Pascal program compiled [140, 141]. In analyzing students' first such compilation, they concluded that just a few types of bugs accounted for a majority of the mistakes in students' programs. They went on to suggest that teaching can most effectively be improved by changing instruction to address and eliminate the high-frequency bugs.

Java Compiler Errors

In more recent work, Jackson et al. similarly recorded compiler errors from custom-built Java IDE [62]. The top ten types of errors represented over half of the total number of occurrences and, interestingly, they also differed from what faculty had believed to be the most common. A unique feature was that any student could access a web page listing their most common errors.

Ahmadzadeh et al. also collected Java compiler errors but using an instrumented compiler [2]. They divided the errors into three categories: syntax errors dealt with the grammar of the language, semantic errors with the meaning of the code being inconsistent, and lexical errors with unrecognized tokens. The observed distribution was 63 % semantic error, 36 % syntax errors, and 1 % lexical errors. Furthermore, only 6 of the 226 distinct semantic errors made up more than half of the error occurrences in each unit. In another experiment, the students were given a debugging task in the form of a faulty program. Students successful at locating bugs were observed using print statements or a filtering approach of commenting out certain lines. Overall, they observed that the majority of good debuggers are good programmers while less than half of the good programmers are good debuggers.

Java Compilation Behavior

Later work dealing with Java has gone beyond examining compilation errors to investigating the complete edit-compile cycle of alternating between editing and compiling a program. In Jadud's work, the BlueJ novice programming environment was modified to, among other metadata, record the time, the source code and, the possible compiler error[62] at every compilation [63]. The frequency distribution of compiler errors by type showed

[62] BlueJ only provides the student with a single error for a compilation even if many exist in the code.

that the minority of different types of compiler errors accounted for the majority of errors. As another observation, overall, it was common to spend very little time before recompiling after an error and instructors observed students often recompiling their programs even without attempting to understand the error. Some students had compiled up to almost 60 times in a single one-hour lab session. Jadud points out that the environment could be modified to guide students past the common errors observed but cautions making this a crutch that students would rely on doing the work for them instead of learning from it.

In another paper, Jadud reports similar observations from students of several first-year programming courses [64]. Furthermore, the paper reports a case study of the session of a single weaker student where he ends up spending a significant amount of time dealing with syntax errors instead of the actual program design task. Overall, Jadud notes that the students exhibited similar behavioral patterns in struggling with the syntax as described by Perkins et al. in solving programming problems in general [110].

Indeed, in this early work from the 80s Perkins et al. discussed some powerful characterizations of the ways how students approach solving a programming problem [110]. They observed two general behavioral patterns that they classified as *stoppers* and *movers*. When faced with difficulties, stoppers will simply stop overwhelmed with the belief that they cannot solve the problem on their own. Movers, on the other hand, will consistently try one thing after another without ever really seeming to be stuck. At the far end of this, *extreme movers* tend to try new fixes with hardly any reflection or apparent convergence to a solution, and end up abandoning promising ideas prematurely or even going in circles trying the same thing over and over.

Going back to Jadud's work, he has also presented an HTML-based visualization of programming sessions in terms of compilation events [64]. Jadud also describes the error quotient (EQ) that aims to quantify how much a student struggles in a programming session based on the encountered compiler errors. Consecutive erroneous compilations add to the quotient and repeating the same error even more so. Overall, Jadud notes that his tools and the EQ allow a teacher to easily view how students are doing when solving the assignments as opposed to simply looking at the end result, and plan interventions when appropriate.

In later work Jadud and Henriksen have published a reimplementation of the BlueJ data collection infrastructure that extends the capabilities by also collecting data about when methods on a class or object are invoked via BlueJ object diagrams [65]. Using this, Tabanao et al., consistent with earlier results, report that top ten error types accounted for 76% of all the compiler errors [144]. Their work has also explored linking students' performance on a programming course to EQ and other compilation data but they have not come up with any real predictive power beyond moderate statistically significant correlation [144, 145].

Rodrigo et al. have also collected similar BlueJ data as Tabanao et al. and supplemented this with human-observed affective states [118]. Continuing this work, Rodrigo and Baker have generated a linear regression model of a student's frustration based on the average number of consecutive compilations with the same edit location, average number of consecutive pairs with the same error, and the average time between compilations but with fairly weak results in terms of predictive power [119].

Fenwick et al. have also collected similar Java compilation data in BlueJ using their own extension called ClockIt [102]. In a survey about the value of insights that this type of data may provide, they found that both the large majority of CS students and faculty perceived it would be helpful for introductory CS students to know about the types of compilation errors encountered and how a student's habits compare to his or her peer. A thing of note compared with much of the other work is that students could also view the few types of graphs visualizing their own activity through a web interface. As for observations, the top five errors recorded were in Jadud's top six and make up over half of all the compiler errors [43]. Their data also indicates that starting early leads to a better grade and that incremental work pays off as well.

In recent work, Utting et al. describe a plan to have a similar data collection feature built into future BlueJ versions[63] in order to collect data about students' behavior on a large scale [148]. They plan to include code-edits on a line-by-line basis, compilation events, and other events such as unit test, debugger, and version control use.

Wittmann et al. have also recorded students' compilation traces, however, with a specific focus on computer graphics programming [151]. They present the SCORE package that records learners' C/C++ code changes between compilations and provides a viewer and analyzer for investigating

[63] The newest versions appear to now include the data collection features since June 2013 at `http://www.bluej.org/`.

code changes. The system is implemented as an Eclipse plug-in that stores the trace locally and students then submit this data along with their project code. The plug-in was used on a university computer graphics course. The authors present an analysis of a single third-year student's trace in two tasks. The results indicate that spatial programming aspects – relating to coordinate changes or transformations – were more difficult for the student than general programming aspects or OpenGL syntax and semantics.

Finally, Retina is a system that not only collects Java code snapshots at compilations – from BlueJ, Eclipse, or a modified compiler – but also makes suggestions to students based on this data via instant messages [100]. The recommendations are based on rules such as suggesting the student work in smaller increments if their rate of errors per compilation is higher than average, or to seek help if they are spending more time on the assignment than expected.

Web-CAT Submissions

Web-CAT is another automated grading system that has been made use of in examining students' progress and behavior in programming assignments. In using Web-CAT students are usually allowed to submit their work for assessment an unlimited number of times before the deadline. Edwards et al. have examined five years of programming assignment submission data from their first three programming courses [41]. In analyzing this data, they were able to find statistically significant results suggesting that when students received higher scores, they had started earlier and finished earlier than on assignments where they received lower scores. They did not appear to spend any more time on their work. The authors suggest a possible explanation to be that when students start earlier, they simply have more opportunities to get help and then go on to perform better.

In other recent work, analyzing a similar large data set from their locally developed Web Submissions System, Falkner and Falkner investigated the relationship of the timeliness of submissions to students' later timeliness of assignment submissions and their average grade from courses [42]. They found that students who submit their first piece of work late seem more at risk of submitting late for the rest of their career and that this behavior also seems to correlate with the grades.

CodeWrite Submissions

Denny and Luxton-Reilly et al. have studied submissions to the CodeWrite web environment where in each exercise students implement a single method [35, 37, 87]. Students write code into a text box, submit the code, and receive immediate feedback. The system is described in more detail in Section 2.4. They have studied the exercise submissions on an introductory Java course with over 400 students.

In the first study, Denny et al. investigated the occurrence of syntax errors [37]. They found that students often struggled with syntax even with the short method implementations. Moreover, students of all levels of ability frequently wrote code that did not compile while weaker students were often unable to fix their syntax errors. Even students in the top quartile of the class were unable to write syntactically correct code in nearly half of the submissions[64].

In a continuation of this study, Denny et al. investigated the types of syntax errors students encounter and the time it takes them to solve those [35]. They confirmed earlier results that a small number of error types account for the most occurrences of errors and the error types were similar to that of previous work. They also found that a lot of time is spent resolving these common errors and that, somewhat surprisingly, more capable students took as long to solve the two most common types of errors as any other students while still far less time is spent with some other types of errors. With this in mind, the authors suggest that targeted interventions around the causes of these errors may yield significant improvements in overall student productivity.

In a third study, Luxton-Reilly et al. investigated the variety of students' correct solutions [87]. They present a taxonomy for classifying solutions with three different hierarchical levels of variation. At the top level, solutions are differentiated by their control flow. Those with identical control flow graphs may be further differentiated by the variation in blocks of code which the authors characterize as a sequence of token classes. Finally, identical sequences of token classes may have been written using a different visual appearance, for example, using different names for identifiers or using whitespace differently. These three levels are called *Structure*, *Syntax*, and *Presentation* in the taxonomy. Finally, the authors present a tool for automatically categorizing method implementations at the struc-

[64] The authors note that this may partly be due to them attempting to solve more challenging exercises.

tural level and report on considerable differences across assignments in the variety of solutions ranging from only a few to even more than 50 different structural approaches.

Marmoset and CVS Repositories

Spacco et al. have collected data using Marmoset, an automated project snapshot and submission system that records snapshots of a student's code to CVS every time the student saves his or her work. [139]. They present the design of a relational database schema for this data.

In a very recent continuation of the Marmoset work, Spacco et al. visualized the number of snapshots being produced relative to the deadline and found that the majority of work was being done 48 hours before the deadline [138]. They report that starting early correlates with better scores. In other very recent work, Balzuweit and Spacco have discussed a prototype web service for visualizing snapshot data like the one Marmoset records [11].

As for other work regarding version control systems, Glassy studied commit patterns to see things like whether students were making incremental progress or waiting to work on the assignment until just before a deadline [49]. Mierle et al. investigated students' code from version control repositories and implemented a system for storing this data in an SQL database [96]. Furthermore, they attempted to find features that would correlate with final course grades but were unable to find any strong predictors, whereas Poncin et al. have applied process mining techniques from business process analysis to investigate software repositories of students' capstone projects [112]. Their work, however, focused on the higher-level processes of software engineering.

Data Mining and Hidden Markov Models

Taking a very different analysis approach, Allevato and Edwards have also applied the data mining technique of frequent episode mining to try to discover frequently occurring patterns in Web-CAT submissions [6]. They found that weaker students had a frequent pattern of removing entire methods from their code. They suggest this to be due to deficiencies not only in the students' implementations but also in their designs which can be more difficult to resolve.

In other recent work, Piech et al. have also used data mining and machine learning techniques to analyze programming session data [111]. They recorded the compilation events in Eclipse. A Karel the Robot assignment using a Karel language based on Java was analyzed more closely. They clustered the many states of code into more high-level milestones and using these modeled each student's development path using a Hidden Markov Model. They further clustered the individual students' HMMs to create a graphical state machine model of the few different high-level development paths students undertook to solve the assignment. The resulting different paths correlated with students' performance and more strongly than the score achieved in the assignment. In other words, how the students solved the assignments predicted their performance better than the mere score they received.

Another line of research that makes use of Hidden Markov Models in capturing the development behavior of students is the work by Kiesmüller et al. [71]. They have studied students' problem solving strategies in the finite state machine -based visual microworld programming environment Kara. Using log data from the system they have built a real-time identifier for previously observed problem solving approaches.

Other Traces

Students' actions and behavior have also been recorded and analyzed in other tasks that relate to programming besides conventional code writing. Blikstein has logged students' actions when writing programs in the NetLogo[65] modeling environment [20]. In exploring sessions using Mathematica scripts and a software tool for viewing an individual trace, he identified the behavior of copy-pasting where the students would switch away from the environment for long periods of time and then suddenly the code would grow notably.

While not based on traces but on human-observation and interviews the study of students' programming in the visual programming environment Scratch needs to be discussed here because of its relevance to the work in this thesis [95]. The authors discovered two patterns of building programs in Scratch: (a) a bottom-up process that starts with the individual Scratch blocks, and (b) a tendency to extremely fine-grained programming. In a bottom-up programming approach one starts with the smallest components, which are then linked together to form a larger subsystem, until a complete top-level system is formed. The students did not then approach the task by thinking on the algorithmic level or the design at all. Fine-grained programming, on the other hand, was when students carried out

[65] http://ccl.northwestern.edu/netlogo/

decomposition until the units became extremely small, and usually lacked logical coherency. The authors note especially disturbing that students avoided the use of the most important structures: conditional execution and bounded loops. They further note that these behaviors are contradictory to the processes students should learn to follow: "(a) to start by designing an algorithm to solve a problem, and (b) to use programming constructs to cleanly structure programs". They suggest that such an exploratory learning environment like Scratch is may then even be detrimental to the study of programming when going forward as bad habits tend to be persistent.

Dealing with the same theme Adams and Webster have examined over 300 student projects made with Alice and Scratch [1]. They found that when creating games in these environments, students used the most variables, `if` statements, and loops. Projects on music videos used far fewer variables and `if` statements. Story-telling projects used the fewest loops, variables, and if statements, but made the most use of dialog. Consequently, depending on the project – gaming projects being the most complex – students may well learn very little about programming concepts if not artificially required to make use of them.

Worth mentioning is also the innovative work on recording and analyzing students simulation sessions in visual algorithm simulation [75] and visual program simulation [132] exercises. In these exercises, learners simulate the workings of an algorithm or, respectively, a program using graphical objects. Traces recorded from the TRAKLA2 system [89] have been used identify students' misconceptions related to heaps [124]. This work has been replicated and expanded using the JSAV [69] algorithm visualization library [68]. Misconceptions related to programming concepts have been studied using traces captured from the UUhistle system [136] for visual program simulation [129].

Finally, in previous work on the js-parsons environment further developed in this thesis, four experts were observed when solving program construction tasks dealing with algorithmic programs. The experts seemed to follow a strategy where they started with control structures. The function signature was first added to the solution, then loops that were reordered if needed, and then the `if`-statements were added, and finally all the remaining lines [61].

Discussion

Various approaches to observing learners' programming sessions in order to study and guide the development process were discussed in this section. Student-led live coding, and video and screen recording provide detailed data but collecting and analyzing data like this takes a lot of effort. Development tools, such as the programming environment, the compiler, or the version control system, and submission systems used in teaching, can be used to capture and analyze data about the programming sessions with much less effort.

Most of the previous work in this area has focused on analyzing the compiler errors students encounter when writing Java. The overall result from these studies is that a few types of errors account for the majority of occurrences – consistent with an early study of students' bugs in Pascal programs. Another observation that emerged from Jadud's work and CodeWrite studies is that students struggle with syntax quite a lot.

A few studies have went beyond counting errors and looked at the Java compilation behavior. Jadud observed frequent compilations and tried to quantify how much a student is struggling based on the compilation behavior. Compilation behavior has also been used to try and predict performance and frustration but there was not much predictive power. CVS data has not yielded notable results in this either. The approach of clustering students' behavior into a few generic development paths which then correlated with performance seems quite promising in this line of research. Also with regard to performance, a study about compilation behavior and a few on students' submissions or code snapshots suggest that, overall, starting work on an assignment early before the deadline leads to a better grade.

An interesting feature in a few of the previous systems that capture and analyze students' programming sessions is that they let students view information about themselves. This is something that could be studied further – whether students make use of and benefit from this type of functionality.

In the work in this thesis, we have recorded and analyzed students' actions in an exercise environment as they are performing code writing and program construction tasks. With regard to previous work, we have collected programming session data that is richer than in any of the previous work: all the code edits and interactive console sessions. Our work also deals with Python as opposed to previous work. As for program construction sessions, our work relates best to the work on collecting data from the somewhat similar novice environments Scratch and Alice, and is a continuation to the initial js-parsons study. The work in this thesis dealing with these themes will be summarized in Chapter 4.

3. Environments for Practicing Programming

In Chapter 1, the following was presented as the first high-level research question of this thesis.

RQ1 *How to minimize the barrier to start practicing programming?*

This was further broken up into three research questions dealing with the specific approaches we have chosen to investigation. Next, we summarize the research in the papers included in this thesis that deal with these questions.

3.1 Web-Based Programming Exercises

As discussed in Chapter 1, the first approach we have studied to lower the barrier to start practicing programming, is the use of web technologies in implementing programming environments for novices. The research question is as follows.

RQ1.1 *Can we, using web technologies, lower the barrier to start practicing programming?*

We have explored this by designing and implementing two Python programming environments for novices. They both reduce the complexity of getting started with practicing programming in the following two ways. First, only a limited set of key functionality is provided in an integrated exercise environment. Second, web technologies are used to improve portability and ease of access.

There is a variety of challenges to implementing a programming environment inside the browser. In the not so far past, it was not at all obvious that such applications could be properly implemented as rich internet applications, as these types of applications are commonly called. Particular challenges include how to implement efficient drawing for a code editor that supports syntax highlighting and the execution environment for a language other than JavaScript which the browser supports. In the first tool we developed, we kind of avoided this as we relied on the Java plug-in that used to be common in browsers. The editor and the execution environment were built on top of Java-based libraries. Subsequently, purely browser-based high performance editors have been developed, such as Ace[1] and CodeMirror[2], as well as some languages are being implemented in JavaScript in order to be run directly in the browser. In the meantime, Java as a web technology lost a lot of its appeal as the installation base decreased and operating systems started treating the plug-in in a hostile manner due to growing concerns for security. In the second tool, we adopted the purely browser-based approach. Each tool is discussed separately in the two sections that follow.

3.1.1 Jype – A Program Visualization and Programming Exercise Tool for Python

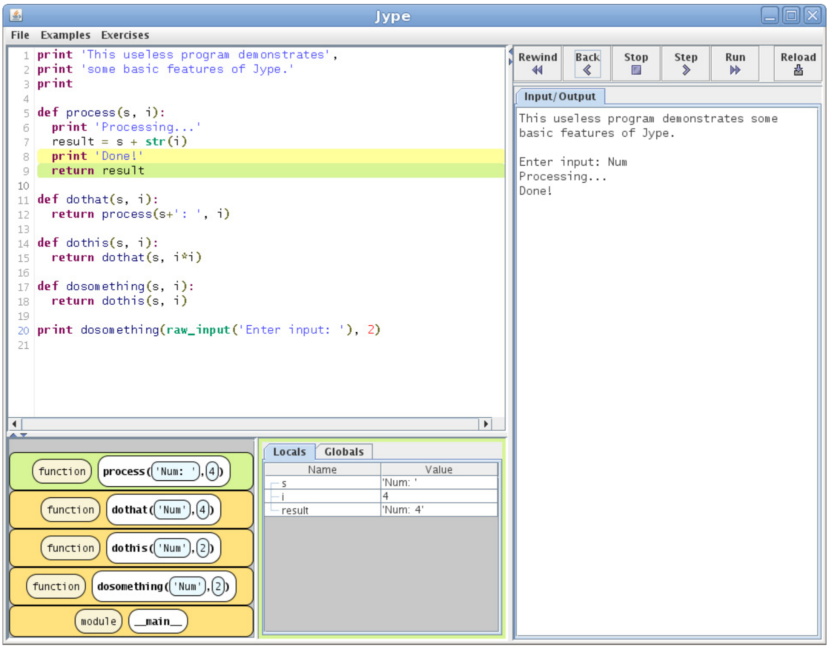
From research into learning programming we derived 5 design goals for a new Python learning tool: 1) facilitate program comprehension, 2) aid in debugging with visualizations and reverse navigation of a program's execution, 3) provide programming exercises with automated feedback, 4) ease of use, and 5) web deployment with a low barrier to entry. In view of these goals, we implemented *Jype*, a web-based programming and program visualization environment that provides automated feedback on programming assignments. The environment is built with Java web technologies and Jython[3], a Python implementation in Java. The system integrates program visualization and automated assessment into the editing and execution environment and provides only a streamlined, limited set of functionality in order to lower the barrier to entry into successfully writing and debugging solutions to programming exercises. Figure 3.1 shows a screenshot of the tool.

The intent of the system is to ease students into programming by letting them quickly build confidence using this simplified environment to implement programs. This is meant to give students a sense of achievement early on and better motivate them to later put the required effort into learning programming and the use of professional tools. In contrast to

[1] http://ace.c9.io/

[2] http://codemirror.net/

[3] http://www.jython.org/



```
print 'This useless program demonstrates',
print 'some basic features of Jype.'
print

def process(s, i):
  print 'Processing...'
  result = s + str(i)
  print 'Done!'
  return result

def dothat(s, i):
  return process(s+': ', i)

def dothis(s, i):
  return dothat(s, i*i)

def dosomething(s, i):
  return dothis(s, i)

print dosomething(raw_input('Enter input: '), 2)
```

```
This useless program demonstrates some
basic features of Jype.

Enter input: Num
Processing...
Done!
```

```
function  process('Num: ',4)
function  dothat('Num',4)
function  dothis('Num',2)
function  dosomething('Num',2)
module  __main__
```

Name	Value
s	'Num: '
i	4
result	'Num: 4'

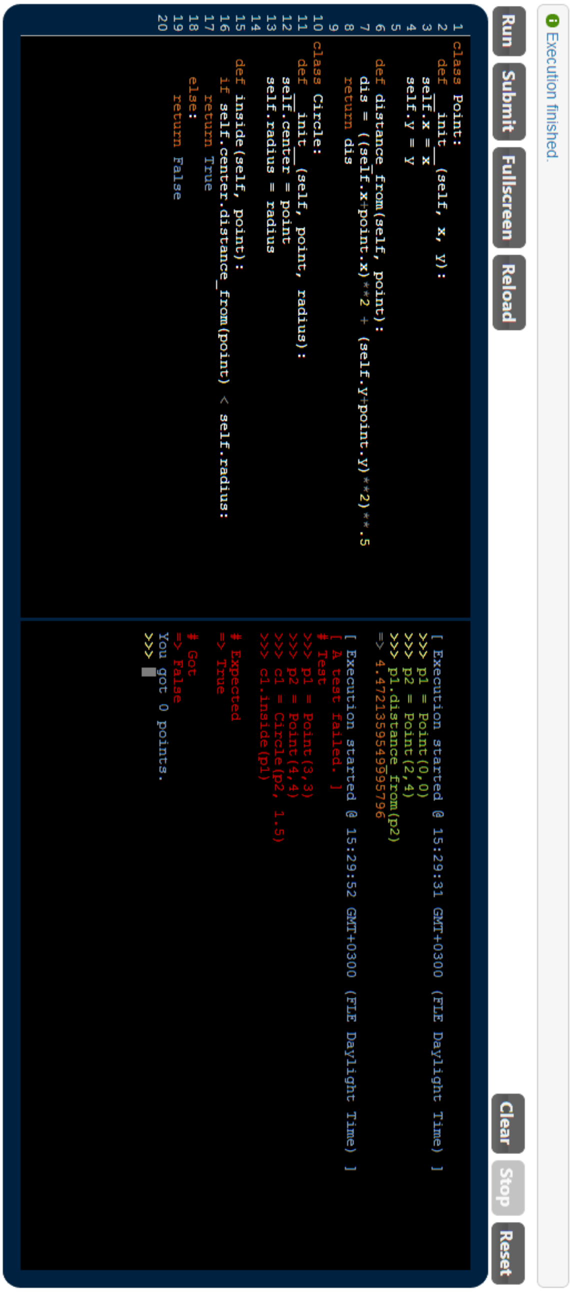
Figure 3.1. A screenshot of the Jype tool as a program is being executed. On the left is the code editor also used to visualize the progress of execution and below that are shown a visualization of the execution stack and variable bindings. On the right is a console for input and output and above that some basic controls.

similar tools, programs are executed client-side.

The system contributes to the research in learning programming by drawing upon existing literature to provide a unique combination of features aimed at supporting novice Python programmers. It was used on an introductory Python course where students could opt to program in it. Student feedback from this pilot study gave some anecdotal evidence that integrating functionality and providing simple web access were found useful. The rationale, design, and implementation of the tool is presented in full in Publication I.

3.1.2 IPPE – In-Browser Python Programming Exercises

The work presented in the previous section was continued later on but the changing landscape of web technologies brought on a shift in the implementation approach. We designed a new tool, abbreviated *IPPE* in this text, that similarly aimed to provide programming exercises with automated feedback, ease of use, and ease of access via web deployment. This new browser-based Python programming environment integrates an editor, an execution environment, and an interactive Python console, and



```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def distance_from(self, point):
    dis = ((self.x+point.x)**2 + (self.y+point.y)**2)**.5
    return dis

class Circle:
  def __init__(self, point, radius):
    self.center = point
    self.radius = radius

  def inside(self, point):
    if self.center.distance_from(point) < self.radius:
      return True
    else:
      return False
```

```
[ Execution started @ 15:29:31 GMT+0300 (FLE Daylight Time) ]
>>> p1 = Point(0,0)
>>> p2 = Point(2,4)
>>> p1.distance_from(p2)
=> 4.47213595499957 96

[ Execution started @ 15:29:52 GMT+0300 (FLE Daylight Time) ]
[ A test failed. ]
# Test
>>> p1 = Point(3,3)
>>> p2 = Point(4,4)
>>> c1 = Circle(p2, 1.5)
>>> c1.inside(p1)

# Expected
=> True

# Got
=> False
You got 0 points.
>>> 
```

Figure 3.2. A screenshot of the IPPE tool. On the left is the code editor and on the right is an interactive console. (image is rotated)

is used to deliver programming assignments with automated feedback. Figure 3.2 shows a screenshot of the tool.

Compared to the earlier system, this one does not provide visualization features but there is a console to enable interactive testing of programs.

Also, this environment is built with HTML and JavaScript only and no Java, thus, further reducing system requirements. Specifically, Python is executed using a CPython Python implementation that has been translated into JavaScript and the code editor uses the CodeMirror library mentioned above. Similar to the earlier system, programs are executed client-side.

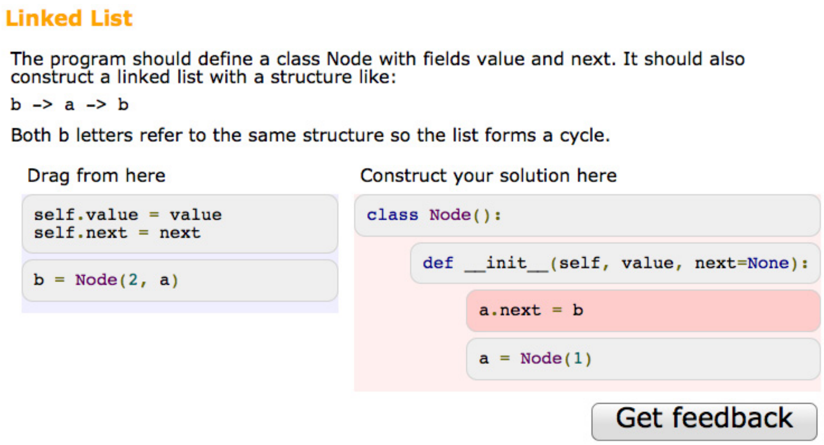
The environment was used on a web development course to deliver three Python programming assignments. Students' perceptions were gauged using a web questionnaire. The reception was generally positive and a great majority of respondents (79 %) agreed that the system should be used on the course again in the future. Free text answers highlighted especially the benefit of there being no setup. The implementation of the environment and students' perceptions of it are presented in full in Publication V. A second focus in this paper was recording and analyzing students' actions and behavior in the environment and this will be discussed in Section 4.2.

3.2 Program Construction Exercises

The previous section mainly dealt with lowering the barrier to start practicing programming by simplifying the tools and the programming environment overall. We have additionally experimented with an approach where the mechanics of describing programs are simplified by letting learners create programs from a given set of building blocks instead of writing statements and expressions from scratch.

As discussed in Section 2.4.1, Ihantola and Karavirta have previously designed and implemented a system called js-parsons for a type of program construction exercises they call two-dimensional Parson's puzzles [61]. In these exercises, learners are given a set of Python code fragments they can use to build the kind of program required of them in the particular assignment. The code fragments are complete lines of code (without indentation) – either a single line or a few lines in a block. As already mentioned in previous chapters, the block structure in Python code is defined by using indentation. In js-parsons, the task then becomes how to place code lines in their proper place both in terms of the order and the indentation in order to put together a program with the expected functionality. Figure 3.3 shows a screenshot of the tool.

In the work in this thesis, we have implemented new features to js-parsons. Particularly, the original version of js-parsons provided only rudimentary automated feedback which we have improved in this work.



Linked List

The program should define a class Node with fields value and next. It should also construct a linked list with a structure like:

```
b -> a -> b
```

Both b letters refer to the same structure so the list forms a cycle.

Drag from here

```
self.value = value
self.next = next
```

```
b = Node(2, a)
```

Construct your solution here

```
class Node():
    def __init__(self, value, next=None):
        a.next = b
        a = Node(1)
```

Figure 3.3. A program construction exercise using js-parsons.

Making use of js-parsons, we have also studied the feasibility of using program construction exercises to practice programming on touch devices. The next two sections summarize the work carried out in this thesis with regard to these two themes.

3.2.1 Automated Feedback in Program Construction Exercises

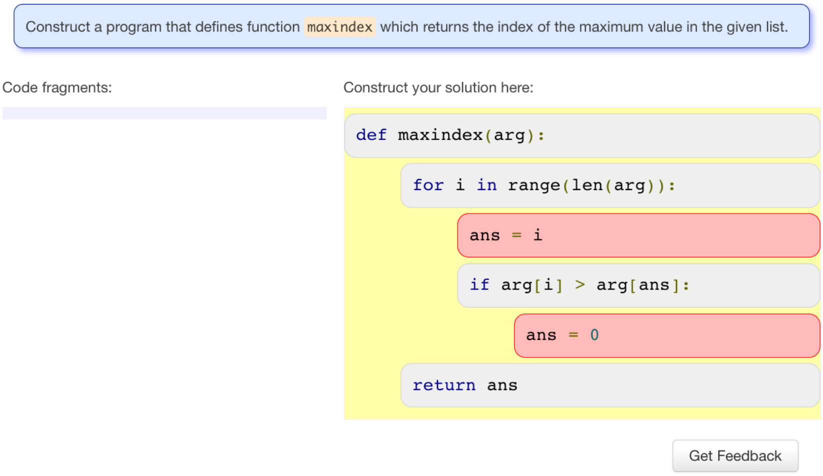
As discussed in Chapter 1, the second approach we have studied to lower the barrier to start practicing programming, is the use of program construction exercises in teaching programming. In experimenting with these exercises, we have encountered a number of issues with the automated feedback provided. From these experiences emerged the following general research question.

> **RQ1.2** *How to improve automated feedback in program construction exercises?*

The original feedback mechanism in js-parsons was based on checking the lines in a constructed program one-by-one starting from the top. When the first line is found that does not belong to that particular row index, it is highlighted. In js-parsons feedback can be requested instantly with a click of a button and by default there is no limitation to the number of such requests. One drawback to this is that with unlimited feedback, this mechanism can be used to build the solution by a simple process of trial-and-error: try to place each of the remaining lines in turn to the last row and request feedback to test if it is correct. However, more importantly, we felt the original type of feedback was inadequate because learners would not get useful feedback on perfectly sensible partial solutions where they had, for example, first placed some control structures, such as loops and function definitions to their correct positions in relation to each others. A better mechanism would simply point out that the partial program is fine but some lines are still missing instead of stating that a line is incorrect just because another line is yet to be added before it. The original type of feedback would in a way immediately guide into thinking about the structure of the program line-by-line in full detail starting from the top. Even with all lines arranged into a faulty program, the original feedback would only point out the first incorrect placement counting from the top, thus, possibly steering the learner towards a fix where this single line is next tried somewhere else instead of trying to figure out the program as a whole. The specific research question is then as follows.

> *How to provide better feedback on partial solutions to program construction exercises?*

In response to this, we implemented a new type of feedback mechanism based on computing longest common subsequences between the single expected solution and the currently constructed program. This feedback points out a minimal number of lines whose repositioning can be used to correct the program if done right. Figure 3.4 shows a screenshot of this type of feedback. The implementation is explained in full in Publication III.



Construct a program that defines function maxindex which returns the index of the maximum value in the given list.

```
def maxindex(arg):
    for i in range(len(arg)):
        ans = i
        if arg[i] > arg[ans]:
            ans = 0
    return ans
```

Figure 3.4. Feedback based on code fragment positions, that is, line-based feedback. One way to correct the program is to reposition the fragments highlighted in red.

Continuing with this theme, in a subsequent analysis of students' use of the js-parsons system with this new type of feedback, we observed some students ending up going in circles in their flawed program design and also using feedback excessively, in a trial-and-error manner. Indeed, even though the new type of line-based feedback we designed is not as directly exploitable to build the solution in the trial-and-error manner described above when discussing the original mechanism, it does still guide the learner towards that single solution quite effectively and is prone to misuse. The study will be discussed in more detail in the next chapter. These observations motivated the following two specific research questions.

How to provide guidance to learners that are going in circles?

How to discourage excessive use of feedback in a trial-and-error manner?

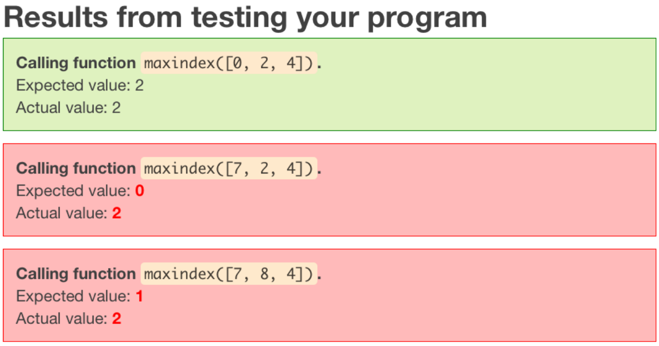
In response to these, first, we implemented a feature where the learner would get a discreet notification when they wound up with a flawed program design that they already had previously constructed. A tab could then be opened that would show them the feedback for the current program and for the the program they previously changed it to. Second, we implemented a penalty for requesting feedback too often. Based on a few simple heuristic rules the feedback functionality would be disabled for a while when it was being used frequently in a way that we deemed trial-and-error behavior. The implementations of these feedback mechanisms are also explained in full in Publication III.

We call the overall feedback mechanism just described line-based feedback. A key weakness with this mechanism is that the feedback is constructed relative to a single correct expected solution which must be unique for all this to work[4]. However, if we could build exercises that would allow alternative solutions, a much larger variety of problems could be supported with these exercises. From a teacher's point of view, it would also be much easier to design exercises without the requirement of a single unique solution. This motivated the following two specific research questions.

Can we provide automated feedback in program construction exercises in a way that does not require a single unique solution?

In response to this, we implemented an alternative form of feedback similar to that traditionally used in programming assignments. The constructed program is run against unit tests and the results of test runs are then presented. Figure 3.5 shows a screenshot of this type of feedback.

[4] The same applies to the pre-existing, original feedback mechanism in js-parsons.



Results from testing your program

Calling function `maxindex([0, 2, 4])`.
Expected value: 2
Actual value: 2

Calling function `maxindex([7, 2, 4])`.
Expected value: 0
Actual value: 2

Calling function `maxindex([7, 8, 4])`.
Expected value: 1
Actual value: 2

Figure 3.5. Execution-based feedback for the program shown in Figure 3.4.

Contrary to what is common in similar browser-based tools, constructed programs are executed client-side. Programs are run using Skulpt[5], a (partial) implementation of Python in JavaScript.

Following the implementation, we went on to experiment with both types of feedback on a CS1 course in Python. Student feedback was collected using a web questionnaire. Overall, attitude towards the exercises was mostly positive. To mention just two questions where there was a high level of agreement, most respondents felt that the exercises helped them to understand other programs (71 %) and that they are worth using on the course again in the future (77 %). Publication IV presents the implementation of execution-based feedback and an experiment carried out to study the effects of the different types of feedback. The experiment is discussed in the next chapter.

3.2.2 Program Construction Exercises on Touch Devices

As discussed in Chapter 1, the third and final approach we have studied to lower the barrier to start practicing programming, is the use of touch devices in order to support mobile and ubiquitous learning. The general research question is as follows.

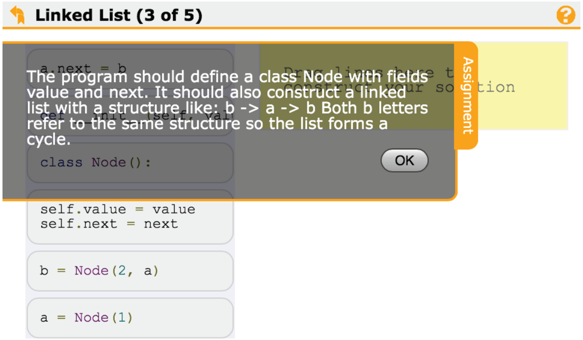
RQ1.3 *How to practice programming on mobile touch devices?*

The program construction exercises discussed in this chapter are small self-contained exercises with a rather simple user interface based on drag-and-dropping code blocks. That is why they seemed well-suited to a mobile context and to touch screen -based devices. This led us to the following specific research question.

[5] `http://www.skulpt.org/`

How to adapt program construction exercises to mobile touch devices?

The mobile application for program construction exercises was built on top of the web-based js-parsons environment with our line-based feedback. However, the user interface was redesigned to make it usable on smaller screens. The interface also resizes automatically making it suitable for both phones and tablets. Furthermore, the PhoneGap[6] library was used to package the web application into a native application that can be installed on the device. The application was packaged for both iOS and Android platforms. Finally, the collection of available exercises is fetched from a remote repository but it is also stored locally to enable offline use. Figure 3.6 shows a screenshot of the tool. The rationale, design, and implementation of the tool is presented in full in Publication III.



The program should define a class Node with fields value and next. It should also construct a linked list with a structure like: b -> a -> b Both b letters refer to the same structure so the list forms a cycle.

```
a.next = b
```

```
class Node():
```

```
self.value = value
self.next = next
```

```
b = Node(2, a)
```

```
a = Node(1)
```

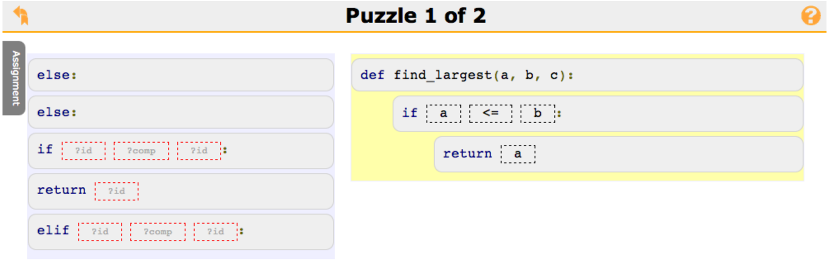
Figure 3.6. A program construction exercise on a mobile device in landscape orientation. The assignment tab is open and shows the description of the exercise. On the left there are gray code fragments that can be used to build the solution onto the yellow area on the right.

In the previous section, execution-based feedback was introduced to enable exercises where there are several acceptable arrangements of the given code fragments, in other words, several alternative solutions. Sometimes, the same set of fragments make a little variation possible but often to allow actual alternative approaches there needs to be additional code available. However, especially on devices with small screens, having a large set of alternative code fragments becomes cumbersome. This led us to the following specific research question.

How to allow a greater variety of different solutions in program construction exercises on touch devices?

[6] `http://phonegap.com/`

In examining solutions to past programming assignments, we discovered variation in the solutions that results from small things, like, from the details how computation is divided into branches in a branch condition. In response to this, we designed and implemented a new type of a program construction exercise where pre-defined parts of the given code fragments may be slightly modified by tapping through a set of given alternative values. For example, a relational operator in an expression could be switched between <, >, <=, and >=. This allows a much greater variety of different types of programs to be built while the interface still remains extremely simple. Figure 3.7 shows a screenshot of the tool.



```
else:
else:
if ?id ?comp ?id:
return ?id
elif ?id ?comp ?id:
```

```
def find_largest(a, b, c):
    if a <= b:
        return a
```

Figure 3.7. A program construction exercise with code fragments that have some parts which can be modified.

An implementation of execution-based feedback on touch devices, similar to that discussed in the previous section, is also presented along with the new exercise type. Specifically, programs are run client-side using the Skulpt library and, consequently, offline use of the application is still possible. Publication VI reviews alternative approaches to practicing programming on mobile devices and describes the rationale, design, and implementation of this new type of an exercise on touch devices.

4. Recording and Analysis of Exercise Sessions

In Chapter 1, the following was presented as the second high-level research question of this thesis.

> **RQ2** *How to, automatically and unobtrusively, record programming exercise sessions and what can we learn from such data?*

We have studied traces of both program construction and programming exercise sessions. Next, we summarize the research in the papers included in this thesis that deal with this question.

4.1 Program Construction Sessions

The program construction exercises based on js-parsons discussed in Section 3.2 provide an opportunity to study the process of creating programs in a restricted setting where learners have a very limited vocabulary for composing programs. Specifically, learners are given only a limited set of code fragments that they can combine to form a program with only few choices left for them in this process: which fragments to use, how to order the fragments, and how to indent the fragments. The js-parsons environment already originally provided support for recording learners' actions as they are solving exercises [61]. In this work, we have made several improvements to the system as discussed in the previous chapter. Accordingly, the recording feature was developed further to account for the new features. Any operations the learner performs on code fragments and any requests for feedback can be recorded as a sequence of events and stored on a remote server. This enables full playback and analysis of a learner's program construction session. Publication II and Publication IV report on analyzing program construction traces collected from university students. The key findings from these two studies are presented next in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 How Do Students Solve Program Construction Exercises?

The recording feature allows us to study how students approach program construction and, on the other hand, in general, to evaluate how they use the js-parsons environment. Consequently, we set off to study methods to visualize and analyze the program construction process. Specifically, we aimed to explore how students arrive at their solution, such as are there common patterns, how much they vary, and how students use automated feedback. The general research question is as follows.

> *Can we identify and quantify students' difficulties and approaches to building programs from automatically recorded traces?*

Data was collected on five assignments on two different programming courses at Aalto University: Web Software Development (WSD) and a CS2 taught with Python. The assignments included short programs of 3-7 code fragments. A student was able to advance to the next assignment only after solving the current one. Analysis focused on the WSD course where around 140 students, almost all of them, solved the assignments.

This study contributes to research in learning programming by presenting a vocabulary and visualization for program construction sessions, as well as identifying some general solution patterns and indicators of poorly proceeding solutions. The key results of the study deal with observations made using a graph-based visualization of program construction sessions. This and other results are discussed in the next three sections. The data collection, analyses, and results are explained in full in Publication II.

Graph Visualization – Common Patterns and Anomalies

A program construction session was conceptualized as a graph where the nodes are different arrangements of code in the student's current (partial) program and the edges represent transitions from one *state* to another as the result of moving a code fragment. Thus a single solution consists of a *path* in the graph from the empty state (no code fragments) to a final state (an arrangement of code that describes an expected program).

First, we examined the following specific research question.

> *Can we identify common patterns in students' approaches to building programs from this data?*

We examined whether any common patterns emerge across solutions by constructing an aggregate graph of all the solution paths for each exercise.

We found there to be a lot of variation in the program construction paths but, when focusing on the edges traversed by most students, two patterns could be identified. Students would generally attempt to solve an exercise either in a linear fashion as if trying to figure out (or having already figured out) the program design line-by-line starting from the top or follow what would seem more structured thinking and start with control structures as if first aiming for a more abstract design of the general flow of the solution. Table 4.1 illustrates how students began the construction process in one of the exercises.

Table 4.1. The percentage of construction sessions where a particular line had been added to the constructed program during the first four steps (S1-S4) in one of the exercises. The numbers show a preference for starting with control flow statements.

Code	S1	S2	S3	S4
`def is_sublist(list1, list2):`	99.3	100.0	100.0	100.0
`  if len(list2) < len(list1):`	0.7	53.7	77.9	86.8
`    list1, list2 = list2, list1`	0.0	4.4	36.0	60.3
`  for i in xrange(len(list2)-len(list1)):`	0.0	35.3	55.1	87.5
`    if list2[i:i+len(list1)] == list1:`	0.0	4.4	14.0	33.8
`      return True`	0.0	1.5	1.5	6.6
`  return False`	0.0	0.7	14.0	19.9

Second, in addition to revealing common patterns we were of course interested in finding out when these patterns break as this would potentially be a sign of difficulties. This led us to examine the following research questions.

Can we identify any anomalies in students' approaches to solving the exercises from this data?

Can we identify struggling students from this data?

To answer these questions, we also went through individual solutions in addition to aggregate graphs. This analysis revealed a relatively common occurrence of going in circles when trying to solve an exercise. What this means is that after repeated changes by moving the code fragments around, the students would end up with an arrangement of code they already had previously constructed. This often seemingly aimless exploration of the problem space is not effective behavior and is a likely indicator of difficulties. Figure 4.1 shows an example of a "loop" in a student's program construction session. Consequently, this observation spurred us to provide additional automated feedback when this occurs as was described in Section 3.2.1 and is fully explained in Publication III.

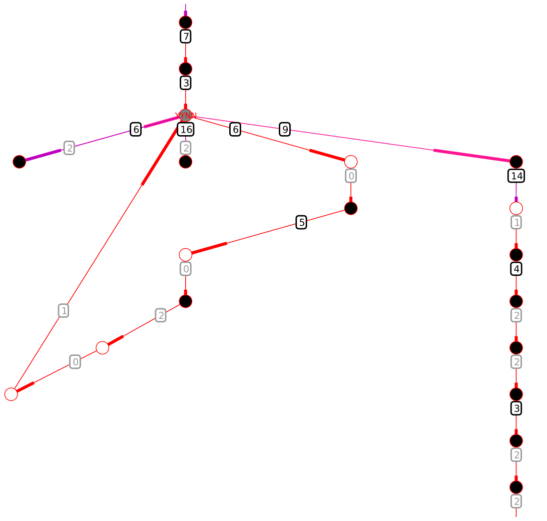


Figure 4.1. Visualization of a part of a program construction session as a graph. Each node represents a single arrangement of student's code. The student has gone in circles with regard to the program design as signified by the loop in the graph (there is some backtracking also). Eventually, the student resorted to trial-and-error. This is seen as a sequence of states where feedback has been requested rapidly in between (the black nodes whose outgoing edges have small numbers in their labels).

Use of Automated Feedback

Additionally, we examined the use of feedback as follows.

> *When and how often do students request feedback?*

In general, students used feedback sparingly and rarely requested it before having added all the code fragments into their program. However, some students were found to have used feedback in a trial-and-error manner where they would request it after every modification they made, with seemingly little thought going into the process. This was evident in the graph visualizations of individual sessions where we also showed how long the student spent time between modifications. There were consecutive sequences of states, where in each state, feedback was requested and there was no pause for thought but instead the code was modified in a matter of only few seconds. Figure 4.1 shows an example of trial-and-error behavior. Following this observation, functionality for limiting too frequent feedback was implemented as was described in Section 3.2.1 and is fully explained in Publication III.

Difficulty of Exercises

Finally, we also examined the use of this data in identifying difficult exercises and code as follows.

Can we gauge the difficulty of particular exercises from this data?

Can we identify problematic parts of code from this data?

Quite naturally, the number of steps needed and the time elapsed when solving an exercise gives a basic estimate on the relative difficulties of the exercises. However, we also explored an alternative method for this. We find that a large variety of states, where feedback has been requested and the arrangement of code has been incorrect, indicates that the exercise has been especially challenging and that students may even have resorted to a bit of guessing. This was true for one of the problems assigned to students where several permutations of the code fragments were among the common incorrect states. Similarly, these states can be used to identify parts of code that students often had trouble with by examining which code fragments were placed incorrectly most commonly. For example, in one of the exercises, object instantiation code was frequently indented inside a wrong block.

4.1.2 Line-Based vs. Execution-Based Feedback

In Section 3.2.1, we discussed the types of automated feedback in program construction exercises – line-based and execution-based. Following the design and implementation of these two types of feedback we went on to empirically examine how the type of feedback affects how students solve an exercise. Program construction exercises with both types of feedback were used on a CS1 Python programming course and students' program construction sessions were recorded in the same manner as in the previous study described above. The students were randomly divided into two groups where each in turn would in two assignments receive execution-based feedback while the other group received line-based feedback. 216 and 165 students in the two groups solved assignment 3.1, the first one of these exercises, giving a total of 381 students taking part in the experiment. The specific research questions are as follows.

Can we identify an effect of different types of feedback on how students solve the exercises from automatically recorded traces?

How does the type of feedback affect students?

Analyses revealed statistically significant differences between the two groups when feedback was different while there were none in the assignments where the type of feedback was the same for both groups. The results show that students who receive execution-based feedback need on average more steps (a median of 16.5 against 8 steps after the first feedback request in assignment 3.1) and in general take longer to solve an exercise (a median of about 37 minutes against about 1 and a half minutes of time elapsed between the first try of solving the assignment 3.1 and eventually completing it). These observations mean that students with line-based feedback were more often able to solve the assignment in one go, with less steps and breaks. On the other hand, execution-based feedback was requested less frequently (in around 38 % against 55 % of states in assignment 3.1) and the respective code was more commonly executable (when feedback was requested around 55 % against 67 % of states that failed to execute). We will discuss these differences more in Chapter 5. The experiment and its results are explained in full in Publication IV.

4.2 Programming Sessions

The work discussed in this chapter so far provides a narrow view of how students approach programming exercises. Certainly, program construction exercises have their limitations to what we can learn from them about how students build programs. Ultimately, most professional software development is about writing and editing code as text. Thus, expanding on the work above we implemented similar recording functionality in a more conventional programming environment – in the browser-based Python programming environment described in Section 3.1.2.

All the actions of a learner in the environment are recorded. Specifically, this trace includes code edits, executing code from both the editor and the console, submissions, and feedback. Similar to the program construction sessions, we capture a timestamped sequence of events of these different types of actions that enables full playback and analysis of individual programming sessions. Web-based tools have also been implemented for viewing and analyzing these traces.

Data was collected on three small assignments delivered to students of a web software development course. The course used the Django[1] library as a server-side web framework and thus included Python programming. The course is intended for third year students but the backgrounds of the students varied greatly. About half were master's and bachelor's level CS majors and the others were from a number of different engineering

[1] https://www.djangoproject.com/

programs. The assignments dealt with functions, strings, lists, basic math, defining a class and methods, and extending a class. An assignment could be submitted an unlimited number of times and the student passed an assignment when all the tests succeeded. 150, 149, and 128 students solved the assignments 1, 2, and 3, respectively.

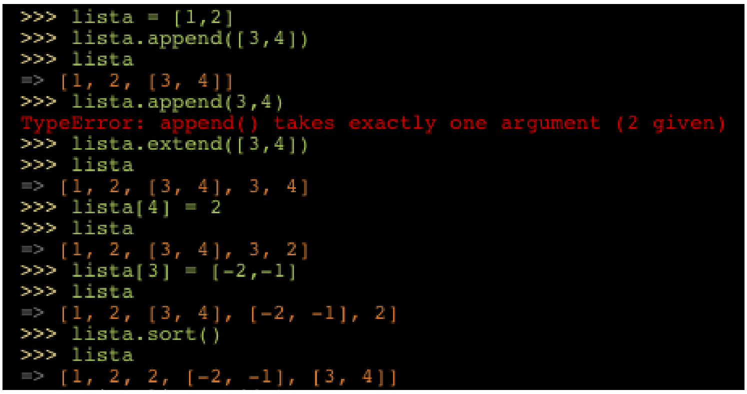
Most similar previous work is on analyzing activity patterns and performance indicators, Java compilation behavior, and submissions as discussed in Section 2.5. In this study, we focused on how students use the interactive console and which kinds of execution errors they run into when developing Python programs. Next, we summarize our findings. The data collection, analyses, and results are explained in full in Publication V.

Console Use

The first specific research question that deals with console use is as follows.

> *Do students use and how do they use the interactive console?*

We manually went through every recorded console interaction. While students had to submit their solution for assessment using the environment, they obviously could also work on it elsewhere. About a third of the students had not used the console at all. Those who used the console, did so for two purposes: interactively testing their program or exploring and experimenting with specific language features and libraries. Figure 4.2 shows an example of the latter case.



```
>>> lista = [1,2]
>>> lista.append([3,4])
>>> lista
=> [1, 2, [3, 4]]
>>> lista.append(3,4)
TypeError: append() takes exactly one argument (2 given)
>>> lista.extend([3,4])
>>> lista
=> [1, 2, [3, 4], 3, 4]
>>> lista[4] = 2
>>> lista
=> [1, 2, [3, 4], 3, 2]
>>> lista[3] = [-2,-1]
>>> lista
=> [1, 2, [3, 4], [-2, -1], 2]
>>> lista.sort()
>>> lista
=> [1, 2, 2, [-2, -1], [3, 4]]
```

Figure 4.2. A console session showing exploration of the functionality of Python lists.

Overall, students used the console actively and productively. For example, over 80 % of those students who used the console in assignment 2, tried all the test runs given in the description of the assignment. About 25 % did more thorough testing with additional test runs that were not provided.

Individual behavior was consistent across assignments.

Another observation possibly related to the console is that students generally used automated feedback rather sparingly. That is, despite that there was no limit to the number of times feedback could be requested, students would ask for feedback up to only a few times in an assignment. On the other hand, we observed many students using the console for testing so it stands to reason that it may have affected their behavior. The tests were given in the form of test runs consisting of a few statements and the value expected as the result of executing these. These tests could then also easily be executed in the console and this might as well have contributed to a lesser need for automated feedback. Thus, we find that it may be beneficial to initially provide students with such tests in order to immediately familiarize them with testing their code themselves instead of relying on the automated feedback.

Execution Errors

The second research question that deals with execution errors is as follows.

> *Which kinds of errors do students encounter in Python programming exercise sessions?*

Despite Python often being praised for its simple and readable syntax, syntax-related exceptions were common (34 % of all exceptions) in students' programming sessions across all the assignments. Most of these resulted from missing colons, empty code blocks after a colon, and missing or unmatched parentheses. Other common exceptions were NameError, AttributeError, and TypeError. Most NameError exceptions resulted from calling functions without importing required packages and from mistyped variable names. Most AttributeError exceptions resulted from running tests given in the assignments before all of the required functionality had been implemented. Most TypeError exceptions resulted from missing a self argument in methods and from a missing or mistyped initialization method. Table 4.2 shows the number and distribution of the different types of errors in each assignment.

Table 4.2. Types of errors encountered by students.

Error	Assignment 1	Assignment 2	Assignment 3
SyntaxError	394 (35%)	195 (17%)	202 (23%)
NameError	274 (24%)	484 (42%)	218 (25%)
IndentationError	173 (15%)	66 (6%)	62 (7%)
TypeError	141 (12%)	252 (22%)	223 (25%)
AttributeError	66 (6%)	135 (12%)	118 (13%)
IndexError	40 (4%)	1 (0%)	25 (3%)
UnboundLocalError	26 (2%)	1 (0%)	15 (2%)
ValueError	21 (2%)	9 (1%)	0 (0%)
ImportError	0 (0%)	15 (1%)	3 (0%)
Other	1 (0%)	7 (1%)	12 (1%)

5. Discussion and Conclusions

In this final chapter, we go back to the research questions laid out in Chapter 1 and review our key results from Chapters 3 and 4 in light of the related work. Each of the research themes in this work is discussed separately following the groupings from those chapters. Finally, we discuss some possible directions for future work.

5.1 Environments for Practicing Programming

In this section, we discuss the first high-level research question of this thesis.

RQ1 *How to minimize the barrier to start practicing programming?*

We have explored this question using three different approaches each of which will be discussed in its own section.

5.1.1 Web-Based Programming Exercises

RQ1.1 *Can we, using web technologies, lower the barrier to start practicing programming?*

We have successfully experimented with using web-based programming environments on programming courses at our university. Specifically, we have designed and implemented two web-based Python environments, referred to as Jype and IPPE, that integrate key functionality needed for practicing programming: assignment delivery, editing and executing code, automated feedback, and program visualization or an interactive console. The environments have been generally well-received and the student feedback seems to support our hypothesis that learners will find this type of simplified and eased access very beneficial. Furthermore, traces of programming sessions showed students making good use of an interactive

Python console that was integrated into the later IPPE environment. A great majority of the respondents to a student questionnaire also agreed that this tool should be used on the course again. Automated feedback in these environments enables self-study anytime and anywhere. Moreover, on account of the web deployment there is no tethering whatsoever to some specific location or computer. Overall, the barrier to start practicing programming is lowered as learners can step right into thinking about programs instead of right from the outset having to also deal with setting up and learning to use more complex tools. We thus conclude that web technologies can be used to implement fully functional novice environments for practicing programming that lower the barrier to entry.

Discussion

In Section 2.4.2 we discussed several similar web-based tools for practicing programming. One key difference of our environments to some, especially the earlier, tools is that they do not provide a proper editor for programming. Instead, a simple text box is provided that, for example, does not support syntax highlighting (e.g. Javala, CodingBat, CodeWrite). Syntax highlighting is a standard feature in any programming environment today and makes it much easier to comprehend code. In Jype, we have used the jEdit programmer's editor and in the IPPE environment the CodeMirror library like in many similarly recent related systems.

Furthermore, many existing tools lack any support for making sense of execution (e.g. CodeLab, CodeSkulptor, CloudCoder). In Jype, programs can be visualized. The IPPE tool, however, also lacks this but we are considering using the existing Online Python Tutor to provide support for this in the future, similar to CS Circles and Runestone Interactive that also make use of this themselves. On the other hand, a rather unique feature in IPPE is the integrated interactive Python console. None of the other web-based novice programming tools have this and traces indicated that students used it actively and productively to test their code and to explore language features and libraries. The CS Circles site provides a possibility to open up a "console" into a separate window where you can try out arbitrary code but it does not actually support true interactive execution with input and output. Moreover, the IPPE tool also stands out as making it possible to log students' behavior in the environment in order to study both the use of the tool and the learning process.

An essential feature in our tools is also that they work with Python.

The very recent CloudCoder tool that appears to have been developed around the same time period as the IPPE tool is in many respects a very similar system. It supports Python, logs students actions, and provides a proper programmer's editor using the Ace library. However, it does no visualization (like IPPE) nor provides a console (unlike IPPE). Finally, CloudCoder executes learners' code on the server using Jython, which was also used client-side in Jype, while the IPPE environment executes code client-side.

Definitely, a key challenge in implementing an environment of this kind is how to provide support for executing code. In this work, we have examined a client-side approach where the learners' code is executed on their own machine. This has several benefits over server-side execution. First, it scales easily because a remote centralized server (or servers) is not being used to execute everyone's code. For example, CloudCoder reportedly ran into performance issues on a large course which prevented deployment [103]. Second, there is less need for secure sandboxing of execution since only the learner's own machine will be affected if something goes wrong. Third, when the environment does not rely on the server for execution, there is not really anything stopping from making it a fully independent tool. This increased mobility allows the tool to be more easily reused and ported from one learning environment to another. Also, the tool could then function offline. In this work, we have not, however, explored offline use. Assignments have been loaded into the practice environments from a remote server and students' submissions have been sent to one too. Moreover, in the IPPE environment, learners' actions were recorded and stored on a remote server.

Client-side execution has its drawbacks as well. Specifically, performance may be an issue. Executing Python using an implementation translated from C to JavaScript turned out to be slow while still manageable. Similarly, executing Python via a Java web plug-in and Jython is slower than native Python. Skulpt JavaScript library that implements some of Python runs much faster but then is only a fairly limited implementation of the language. Additionally, in the environments we have designed, automated assessment is also performed in the client-side execution environment. What this means is that there is a real risk of a student tapping into the assessment routines and falsifying their results. For this reason, you may need to think twice about using these scores for grading. However, in our assignments, the student would still not gain access to a solution (only the tests) and they would have to submit code along the results of assessment. Therefore, the scores could also be verified at the end of the course as a batch run on the students' submitted code. Thus, fear of getting caught doing this and the resulting disciplinary actions should be deterrent enough.

Overall, one might critique against the systems presented on the basis that, after all, everyone has to eventually deal with the real tools of the trade so why postpone this with these temporary crutches. This is a reasonable opinion and concerns a novice environment of any kind. However, we argue that while some students may have no trouble putting in the required effort to get up and running, and into practicing the actual skills, others will initially simply lack the required motivation. This may be the case for non-majors in particular. Getting the computer to perform stuff as you instructed is satisfying and even fun, and students should get a sense of this enjoyment early on to motivate them to persevere through all the complexities. Moreover, we should not get hung up on this but remember that there are lots of other potential advantages to a web-based environment as discussed in length in Section 1.5.

5.1.2 Program Construction Exercises

The second approach we have experimented with is the use of program construction exercises in teaching. These exercises aim to lower the barrier to practicing programming by simplifying how programs are described. Specifically, in the type of exercises we have explored, programs are constructed by drag-and-dropping fragments of Python code in a web-based environment. The exercises were described in more detail in Section 3.2. We have used program construction exercises on several programming courses and they have been met with a mostly positive response. In fact, most respondents to a student questionnaire agreed that these exercises helped them to understand other programs and that they should be used again on the CS1 course they were taking. We conclude that js-parsons program construction exercises are a useful supplemental teaching method that students, in general, seem to value.

RQ1.2 *How to improve automated feedback in program construction exercises?*

In particular, in this work, we have studied the provision of automated feedback in program construction exercises. We have argued that the original implementation of feedback in js-parsons did not support constructing programs in a structured way. If used on an overall blueprint of a program containing control flow statements, it would guide towards thinking about the details line-by-line starting from the top. Additionally, with unlimited feedback, this approach lent itself too easily to a trial-and-error approach. Consequently, we implemented a new type of feedback mechanism that points out a minimal number of lines whose repositioning can be used to correct the program if done right. In experimenting with this type of feedback, we found that students still quite often seemed to resort to trial-and-error use of it if they were initially unable to solve the problem correctly. This behavior should be discouraged and instead students should seek to understand what the problem is. Thus, we implemented a heuristic rule for limiting the use of feedback.

Additionally, we observed students going in circles in their design as if totally unaware of what they were doing. To let students know about how little they were making progress as a hint that they should stop and think, we implemented a discreet notification to let them know when this occurs.

Both these new feedback mechanisms, especially the latter, rely on knowledge of what has been done during the exercise session, that is, the recorded interaction trace. They are somewhat novel in this sense. We have yet to formally experiment with the use of these new mechanisms but we conclude that this data-driven approach of improving feedback, based on what has been seen in recorded traces, has good potential to improve the effectiveness of these types of educational tools.

We have additionally implemented an alternative form of automated feedback based on running tests on the constructed program. We call this execution-based feedback and the one just discussed line-based feedback. In our line-based approach, feedback is constructed relative to a single unique solution. Execution-based feedback enables alternative correct solutions. This type of feedback is discussed more below in relation to traces recorded in order to learn how the two types of feedback affect students.

RQ1.3 *How to practice programming on mobile touch devices?*

In addition to web-based environments, we have studied how to make use of touch screen based mobile devices in practicing programming and implemented an application for this. We have argued that program construction exercises are an excellent fit to this context especially because of their simple user interface while programs may still be constructed using a general-purpose language. Independent of our work, Pritchard and Vasiga have also suggested this type of use for their similar type of an exercise they call code scrambles: “Can a useful programming course be taught on a touch-screen/tablet/iPad? . . . code scrambles might work even better on a touchscreen.” [114]. The key weakness is that this type of exercise greatly limits the freedom of expression in choosing the program design. In order to alleviate this, we have also introduced a type of exercise that lets some pre-assigned parts of the code be modified by tapping through alternatives. We conclude that program construction exercises have great potential in enabling programming to be practiced on mobile touch devices and thus in facilitating ubiquitous learning.

Discussion

Several related systems have previously been designed that change the mechanics of how programs are described as discussed in Section 2.4.1. Many of these are also based on constructing a program from programming blocks of some kind. The key difference to these is that in the program construction exercises as we define them, the blocks are made up of actual written code in a general-purpose language instead of replacing parts of it with visual metaphors (e.g. JPie [51], the first version of Alice [27]) and more typically entirely with a special-purpose teaching language (e.g. Snap![1], Alice 3[2] [30], Scratch[3] [90]). For us, the language is part of what students are to learn instead of simply being a vehicle for teaching the basic concepts. Therefore, we have explored simplifying the use of the particular programming language instead of replacing it with a different notation. We expect this practice then to result in good transfer of knowledge when going forward in learning to program with that language. Similarly, in the mobile context, while TouchDevelop provides a powerful programming environment it does so by using a menu-driven language specially designed for this purpose.

Moreover, we have explored supporting self-study with appropriate formative automated feedback. Apart from the other tools for program construction exercises, similar environments support learning by experimentation, tinkering, and open-ended exploration well, for example, by letting learners program animations like in Scratch and Alice. These tools do not provide a proper framework for assignments with automated guidance

[1] `http://snap.berkeley.edu/`

[2] `http://www.alice.org/`

[3] `http://scratch.mit.edu/`

to complete them independently. Furthermore, as discussed in Section 2.5.2, students working in such exploratory environments have been seen to adopt bad habits of programming [95] and to not make enough use of or even avoid the use of important basic programming concepts like conditional execution, thus, lessening the value of this practice [1, 95].

However, we have not attempted to measure the specific impact of program construction exercises on learning. In fact, the main purpose for our use of these and any kind of automated tools is to make up for a lack of individual guidance. To really put it bluntly, in this context, we have succeeded if we have not made things worse and it appears students mostly valued these types of exercises. However, we must note that the questionnaire used as a basis for evaluating how students perceived program construction exercises was answered by only 163 students (37 %). There may be bias in any direction in that sample. Drawing upon our experiences with these exercises in a web environment we also argue that learners will find them an attractive learning method on mobile touch devices but we have yet to evaluate this.

In their survey of literature on the teaching of introductory programming, Pears et al. discuss the issue of why so few teaching tools have seen wide adoption [109]. They cite missing support for modifying the tool for a different context as one reason, but more importantly the current research process and funding that does not support advancing a tool beyond a research prototype unusable by anyone other than the developers themselves. Neither is our work without blame in this regard. However, something that speaks for the concept and the js-parsons implementation of program construction exercises that have both been developed further in this work, is that they have gained some wider adoption. Exercises using js-parsons are being used as part of an interactive web-based book on Python programming[4] and, based on some email we have received, there are others making use of them, too.

Finally, the new feedback mechanisms about excessive feedback and going in circles have yet to be piloted on a programming course. Similarly, as mentioned above, we have not yet conducted formal experiments on the use of our applications for mobile touch devices.

[4] http://interactivepython.org/runestone/static/thinkcspy/index.html

5.2 Recording and Analysis of Exercise Sessions

In this section, we discuss the second high-level research question of this thesis.

> **RQ2** *How to, automatically and unobtrusively, record programming exercise sessions and what can we learn from such data?*

We have explored recording program construction and programming exercise sessions as discussed next. Overall, the approach lets us study patterns and difficulties in programming in a quantifiable way with little effort required in the process of collecting data.

5.2.1 Program Construction Sessions

We have used automated recording of exercise sessions to learn about how students solve program construction tasks. An important observation is that the data is suitable for revealing difficulties that are not to be seen in the end result of the task. Students are almost without fail able to solve the exercises with the help of automated feedback but there is great variance in the process of how this is accomplished. We presented a graph-based visualization of program construction sessions and used it to identify two overall approaches to program construction: line-by-line and control structures first. We also observed behavior that was indicative of difficulties in solving the exercise: backtracking, going in circles, and excessive, trial-and-error use of feedback. Furthermore, we used these types of traces to discover that the type of automated feedback has a significant effect on how students solve the exercises. We conclude that automatically recorded program construction sessions enable to reveal information about the learning process that can be used plan interventions and in improving learning content and tools.

Discussion

We found a structured pattern of constructing programs starting with control structures to be common in the traces. Yet, students in a somewhat related context – programming in Scratch — have been seen to exhibit ineffective behaviors, such as bottom-up programming as discussed in Section 2.5.2. However, our data came from more advanced students than novices – mainly from participants of a Web Software Development course intended for third year students. True novices could reasonably have entirely different kinds of patterns. We have collected traces from a CS1 course but have yet to properly analyze them in this respect.

The students who exhibited behaviors, such as going in circles and trial-and-error use of feedback seem very similar to the extreme movers characterized by Perkins [110] and discussed in Section 2.5.2. They keep on making changes without too much reflection or any apparent convergence to a solution. However, while we can see this type of behavior indicative of difficulties, we have not addressed the question of what the underlying problem is nor is it generally possible to do using this type of data alone. Even though we can see a student making circles in their program design, it is still difficult to make clear conclusions as to what is causing this apparent confusion. Is there maybe a specific construct whose meaning they are unable to grasp?

Finally, there was a significant difference in how students solved program construction exercises with different types of feedback. The results were presented in Section 4.1.2. It would seem that the execution-based feedback provided less guidance and the exercise sessions were then longer both in terms of steps and time. There was also a statistically significant difference in the number of sessions in an exercise. With execution-based feedback students more often abandoned the task and tried again at a later time. This is not entirely surprising because the line-based feedback gives direct suggestions on where the learner can start about fixing the program. This feedback, in a way, treats the exercise as a puzzle and gives a suggestion which pieces could be moved to complete it. With enough tries the learner should eventually always be able to reach a solution. On the other hand, the usefulness of execution-based feedback relies on learners' ability to trace the execution of their program and this way track down the cause of incorrect behavior – and they may be unable to do this without additional study of the programming constructs involved. In fact, instead of relying on students' ability to do this without support we could provide additional feedback on the execution of tests as program visualizations similar to the what is done in the code sorting exercises in the ViLLE environment. We could also consider combining these two types of feedback: initially give execution-based feedback but also in some cases in a limited way provide line-based feedback to hint where to start searching for the problem.

5.2.2 Programming Sessions

We have, as well, demonstrated the automated collection of detailed traces of programming sessions in a web-based programming environment. The recorded traces let teachers and researchers view a full playback of the programming session in the environment. We demonstrated the use of this data in analyzing how students use an interactive console and what kinds of execution errors they encounter in Python programming. Students made use of the console both for testing their code and for exploring language features. A variety of error types were observed while, consistent with previous studies, only a minority of those accounted for the majority of occurrences. We conclude that automatically recorded programming sessions enable to reveal information about the learning process that can be used plan interventions and in improving learning content and tools.

Discussion

In Section 2.5.2 we discussed several similar endeavors to study how students approach the solving of programming tasks. A lot of the work has dealt with submissions to automated assessment systems (e.g. WEB-Cat, CodeWrite) or Java and its compiler errors and learners' compilation behavior. We have explored Python. Moreover, previously, little work has been done to record and study entire programming sessions. Very recently, other tools have, apparently around the same time period as our work, implemented support for recording such traces (BlueJ [148], CloudCoder [103]). We have additionally gone on to capture traces from students. These gave us the unique opportunity to study how students made use of the interactive console provided for them in the IPPE environment. Indeed, we imagine a lot can be learned from these types of traces and, eventually, we may be able to recognize patterns and difficulties during the session, thus, providing new opportunities for automated feedback.

From one point of view, we have tried to learn about the students' programming processes unobtrusively, that is, without intervening. However, any kind of system that is not a natural part of the learning process is an intervention and the IPPE environment can be regarded as such, as well. This should be considered when evaluating the results. For example, students may not have used the console so much had it not been placed so centrally next to the editor.

Furthermore, the traces collected and the questionnaire data on the use of the IPPE environment came from students of a Web Software Development course which is not intended for complete novices. This should be considered when evaluating their responses. Indeed, there might be some issues in using the system that these more advanced students did not stumble upon. On the other hand, the fact that even more advanced students found the IPPE environment adequate enough for a few programming exercises may be considered a praise.

Finally, with regard to the traces, we only have detailed data from the self-selected group of students who opted to use the environment for more than simply submitting the exercises. However, we found no correlation between choosing to use the environment and students' performance on the course, so it would seem the sample did not differ from the population of the class in this respect at least.

5.3 Future Work

There are several possible lines of future research.

The IPPE environment opens up endless possibilities to replicate studies about students' programming habits, as well as to learn about any difficulties in a quantifiable way. Those can then be addressed in teaching in a prioritized manner. Moreover, similar to some previous work discussed in Section 2.5 (e.g. [62] and [102]), details about the process could be given back to the students themselves to let students evaluate their working habits in terms of their peers and go on to improve themselves independently.

One weakness with the IPPE tool is that contrary to the Jype tool and typical IDEs it does not provide a visual debugger. One possibility would be to explore providing program visualizations using the previously-existing Online Python Tutor tool. Similarly, this tool could also be used to complement execution-based feedback in program construction exercises.

As for the mobile applications, we have explored new ways to practice programming on touch devices but next their effectiveness must be formally evaluated.

Finally, so far, we have only explored using program construction exercises in Python but are working on implementations for other languages.

Bibliography

[1] Joel C. Adams and Andrew R. Webster. What do students learn about programming from game, music video, and storytelling projects? In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 643–648. ACM, 2012.

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An Analysis of Patterns of Debugging Among Novice Computer Science Students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 84–88. ACM, 2005.

[3] Tuukka Ahoniemi and Tommi Reinikainen. ALOHA – A Grading Tool for Semi-automatic Assessment of Mass Programming Courses. In *Proceedings of the 6th Baltic Sea conference on Computing Education Research*, Koli Calling '06, pages 139–140. ACM, 2006.

[4] Kirsti Ala-Mutka and Hannu-Matti Jarvinen. Assessment Process for Programming Assignments. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, ICALT '04, pages 181–185. IEEE, 2004.

[5] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: A Lightweight Pedagogic Environment for Java. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 137–141. ACM, 2002.

[6] Anthony Allevato and Stephen H. Edwards. Discovering Patterns in Student Activity on Programming Assignments. In *2010 ASEE Southeastern Section Annual Conference and Meeting*, 2010.

[7] Carl Alphonce and Blake Martin. Green: A Customizable UML Class Diagram Plug-in for Eclipse. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 168–169. ACM, 2005.

[8] Carl Alphonce and Phil Ventura. Object Orientation in CS1-CS2 by Design. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '02, pages 70–74. ACM, 2002.

[9] Carl Alphonce and Phil Ventura. QuickUML: A Tool to Support Iterative Design and Code Development. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 80–81. ACM, 2003.

[10] John R. Anderson and Brian J. Reiser. The LISP Tutor. *BYTE*, 10(4):159–175, 1985.

[11] Evan Balzuweit and Jaime Spacco. SnapViz: Visualizing Programming Assignment Snapshots. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 350–350. ACM, 2013.

[12] Mordechai Ben-Ari. Constructivism in Computer Science Education. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '98, pages 257–261. ACM, 1998.

[13] Mordechai Ben-Ari. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[14] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. A Decade of Research and Development on Program Animation: The Jeliot Experience. *Journal of Visual Languages and Computing*, 22(5):375–384, 2011.

[15] Jens Bennedsen and Michael E. Caspersen. Revealing the Programming Process. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 186–190. ACM, 2005.

[16] Jens Bennedsen and Michael E. Caspersen. Failure Rates in Introductory Programming. *SIGCSE Bulletin*, 39(2):32–36, 2007.

[17] Joe Bergin. Karel Universe Drag & Drop Editor. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '06, pages 307–307. ACM, 2006.

[18] Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis. *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. Dream Songs Press, 2005.

[19] David Binkley. Source Code Analysis: A Road Map. In *Future of Software Engineering*, FOSE '07, pages 104–119. IEEE, 2007.

[20] Paulo Blikstein. Using Learning Analytics to Assess Students' Behavior in Open-ended Programming Tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pages 110–116. ACM, 2011.

[21] Duane Buck and David J. Stucki. JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 16–20. ACM, 2001.

[22] Ben A. Calloni and Donald J. Bagert. Iconic Programming in BACCII vs. Textual Programming: Which is a better learning environment? In *Proceedings of the Twenty-fifth SIGCSE Symposium on Computer Science Education*, SIGCSE '94, pages 188–192. ACM, 1994.

[23] Ben A. Calloni and Donald J. Bagert. Iconic Programming for Teaching the First Year Programming Sequence. In *Proceedings of the Frontiers in Education Conference*, FIE '95. IEEE, 1995.

[24] Ben A. Calloni, Donald J. Bagert, and H. Paul Haiduk. Iconic Programming Proves Effective for Teaching the First Year Programming Sequence. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 262–266. ACM, 1997.

[25] Martin C. Carlisle. Raptor: A Visual Programming Environment for Teaching Object-oriented Programming. *Journal of Computing Sciences in Colleges*, 24(4):275–281, 2009.

[26] Trevor D. Collins and Pat Fung. A Visual Programming Approach for Teaching Cognitive Modelling. *Computers & Education*, 39(1):1–18, 2002.

[27] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: A 3-D Tool for Introductory Programming Concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116, 2000.

[28] James H. Cross II and T. Dean Hendrix. jGRASP: A Lightweight IDE with Dynamic Object Viewers for CS1 and CS2. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '06, pages 356–356. ACM, 2006.

[29] Garrett Dancik and Amruth Kumar. A Tutor for Counter-Controlled Loop Concepts and Its Evaluation. In *Proceedings of the 33rd Frontiers in Education Conference*, FIE '03, pages T3C/7–T3C/12. IEEE, 2003.

[30] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. Mediated Transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 141–146. ACM, 2012.

[31] Leliane Nunes de Barros, Ana Paula dos Santos Mota, Karina Valdivia Delgado, and Patricia Megumi Matsumoto. A Tool for Programming Learning with Pedagogical Patterns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 125–129. ACM, 2005.

[32] Ezequiel Denegri, Guillermo Frontera, Antonio Gavilanes, and Pedro J. Martín. A Tool for Teaching Interactions Between Design Patterns. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 371–371. ACM, 2008.

[33] Paul Denny, Diana Cukierman, Andrew Luxton-Reilly, and Ewan Tempero. A Case Study of Multi-Institutional Contributing-Student Pedagogy. *Computer Science Education*, 22(4):389–411, 2012.

[34] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 113–124. ACM, 2008.

[35] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All Syntax Errors Are Not Equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80. ACM, 2012.

[36] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. CodeWrite: Supporting Student-driven Practice of Java. In *Proceedings*

of the 42nd ACM Technical Symposium on Computer Science Education, SIGCSE '11, pages 471–476. ACM, 2011.

[37] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the Syntax Barrier for Novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 208–212. ACM, 2011.

[38] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[39] Benedict du Boulay. Some Difficulties of Learning to Program. In James C Spohrer and Elliot Soloway, editors, *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, 1989.

[40] Benedict du Boulay, Tim O'Shea, and John Monk. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man–Machine Studies*, 14(3):237–249, 1981.

[41] Stephen H. Edwards, Jason Snyder, Manuel A. Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. Comparing Effective and Ineffective Behaviors of Student Programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14. ACM, 2009.

[42] Nickolas J.G. Falkner and Katrina E. Falkner. A Fast Measure for Identifying At-Risk Students in Computer Science. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 55–62. ACM, 2012.

[43] James B. Fenwick Jr., Cindy Norris, Frank E. Barry, Josh Rountree, Cole J. Spicer, and Scott D. Cheek. Another Look at the Behaviors of Novice Programmers. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 296–300. ACM, 2009.

[44] Eric Fernandes and Amruth N. Kumar. A Tutor on Scope for the Programming Languages Course. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 90–93. ACM, 2004.

[45] Sally Fincher and Marian Petre. *Computer Science Education Research*. Psychology Press, 2004.

[46] Rex E. Gantenbein. Programming as Process: A "Novel" Approach to Teaching Programming. In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '89, pages 22–26. ACM, 1989.

[47] Stuart Garner. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Journal of Issues in Informing Science and Information Technology*, 4:491–501, 2007.

[48] Alessio Gaspar and Sarah Langevin. Restoring "Coding with Intention" in Introductory Programming Courses. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education*, SIGITE '07, pages 91–98. ACM, 2007.

[49] Louis Glassy. Using Version Control to Observe Student Software Development Processes. *Journal of Computing Sciences in Colleges*, 21(3):99–106, 2006.

[50] Ephraim P. Glinert. Towards 'Second Generation' Interactive, Graphical Programming Environments. In *Proceedings of the IEEE Workshop on Visual Languages*, VL '86, pages 61–70. IEEE, 1986.

[51] Kenneth J. Goldman. A Concepts-First Introduction to Computer Science. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 432–436. ACM, 2004.

[52] Kathryn E. Gray and Matthew Flatt. ProfessorJ: A Gradual Introduction to Java Through Language Levels. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 170–177. ACM, 2003.

[53] Wayne D. Gray and John R. Anderson. Change-Episodes in Coding: When and how do programmers change their code? In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 185–197. Ablex Publishing, 1987.

[54] David Gries. What should we teach in an introductory programming course? In *Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '74, pages 81–89. ACM, 1974.

[55] Philip J. Guo. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584. ACM, 2013.

[56] Marc Hesenius, CarlosDario Orozco Medina, and Dominikus Herzberg. Touching Factor: Software Development on Tablets. In Thomas Gschwind, Flavio Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2012.

[57] Christopher D. Hundhausen, Jonathan L. Brown, Sean Farley, and Daniel Skarpas. A Methodology for Analyzing the Temporal Evolution of Novice Programs Based on Semantic Components. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 59–71. ACM, 2006.

[58] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[59] Wolfgang Hürst, Tobias Lauer, and Eveline Nold. A Study of Algorithm Animations on Mobile Devices. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 160–164. ACM, 2007.

[60] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 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, pages 86–93. ACM, 2010.

[61] Petri Ihantola and Ville Karavirta. Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. *Journal of Information Technology Education: Innovations in Practice*, 10:1–14, 2011.

[62] James Jackson, Michael Cobb, and Curtis Carver. Identifying Top Java Errors for Novice Programmers. In *Proceedings of the 35th Annual Frontiers in Education Conference*, FIE '05, pages T4C/24–T4C/27. IEEE, 2005.

[63] Matthew C. Jadud. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[64] Matthew C. Jadud. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84. ACM, 2006.

[65] Matthew C. Jadud and Poul Henriksen. Flexible, Reusable Tools for Studying Novice Programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pages 37–42. ACM, 2009.

[66] Jhilmil Jain, James H. Cross, II, T. Dean Hendrix, and Larry A. Barowski. Experimental Evaluation of Animated-Verifying Object Viewers for Java. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, pages 27–36. ACM, 2006.

[67] Ville Karavirta, Ari Korhonen, Lauri Malmi, and Thomas Naps. A Comprehensive Taxonomy of Algorithm Animation Languages. *Journal of Visual Languages and Computing*, 21(1):1–22, 2010.

[68] Ville Karavirta, Ari Korhonen, and Otto Seppala. Misconceptions in Visual Algorithm Simulation Revisited: On UI's Effect on Student Performance, Attitudes and Misconceptions. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*, LaTiCE '13, pages 62–69. IEEE, 2013.

[69] Ville Karavirta and Clifford A. Shaffer. JSAV: The JavaScript Algorithm Visualization Library. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 159–164. ACM, 2013.

[70] Caitlin Kelleher and Randy Pausch. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

[71] Ulrich Kiesmueller, Sebastian Sossalla, Torsten Brinda, and Korbinian Riedhammer. Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 274–278. ACM, 2010.

[72] Päivi Kinnunen and Lauri Malmi. Why students drop out CS1 course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 97–108. ACM, 2006.

[73] Michael Kölling. The Greenfoot Programming Environment. *Transactions on Computing Education*, 10(4):14:1–14:21, 2010.

[74] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ System and Its Pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[75] Ari Korhonen. *Visual Algorithm Simulation*. Doctoral dissertation, Helsinki University of Technology, 2003.

[76] Amruth Kumar. Dynamically Generating Problems on Static Scope. In *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '00, pages 9–12. ACM, 2000.

[77] Amruth Kumar. A Scalable Solution for Adaptive Problem Sequencing and Its Evaluation. In *Adaptive Hypermedia and Adaptive Web-Based Systems*, volume 4018 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2006.

[78] Amruth N. Kumar. Learning the Interaction Between Pointers and Scope in C++. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 45–48. ACM, 2001.

[79] Amruth N. Kumar. Results from the Evaluation of the Effectiveness of an Online Tutor on Expression Evaluation. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 216–220. ACM, 2005.

[80] Amruth N. Kumar. Explanation of Step-by-Step Execution as Feedback for Problems on Program Analysis, and Its Generation in Model-Based Problem-Solving Tutors. *Technology, Instruction, Cognition and Learning Journal*, 4(1), 2006.

[81] Mikko-Jussi Laakso, Erkki Kaila, Teemu Rajala, and Tapio Salakoski. Define and Visualize Your First Programming Language. In *Proceedings of the Eighth IEEE International Conference on Advanced Learning Technologies*, ICALT '08, pages 324–326. IEEE, 2008.

[82] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 14–18. ACM, 2005.

[83] Timo Lehtonen. Javala – Addictive E-Learning of the Java Programming Language. In *Proceedings of the 5th Annual Finnish / Baltic Sea Conference on Computer Science Education*, Koli Calling '05, pages 41–48. University of Joensuu, 2005.

[84] Raymond Lister. Teaching Java First: Experiments with a Pigs-early Pedagogy. In *Proceedings of the Sixth Australasian Conference on Computing Education*, ACE '04, pages 177–183. Australian Computer Society, 2004.

[85] Ju Long. Just For Fun: Using Programming Games in Software Programming Training and Education – A Field Study of IBM Robocode Community. *Journal of Information Technology Education*, 6:279–290, 2007.

[86] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships Between Reading, Tracing and Writing Skills in Introductory

Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, pages 101–112. ACM, 2008.

[87] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. On the Differences Between Correct Student Solutions. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 177–182. ACM, 2013.

[88] I. Scott MacKenzie and R. William Soukoreff. Text Entry for Mobile Computing: Models and Methods, Theory and Practice. *Human–Computer Interaction*, 17(2-3):147–198, 2002.

[89] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[90] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *Transactions on Computing Education*, 10(4):16:1–16:15, 2010.

[91] Raina Mason and Graham Cooper. Mindstorms Robots and the Application of Cognitive Load Theory in Introductory Programming. *Computer Science Education*, 23(4):296–314, 2013.

[92] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[93] Sean McDirmid. Coding at the Speed of Touch. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11, pages 61–76. ACM, 2011.

[94] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning Computer Science Concepts with Scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pages 69–76. ACM, 2010.

[95] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of Programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 168–172. ACM, 2011.

[96] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. Mining Student CVS Repositories for Performance Indicators. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5. ACM, 2005.

[97] Bradley N. Miller and David L. Ranum. Beyond PDF and ePub: Toward an Interactive Textbook. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 150–155. ACM, 2012.

[98] Antonija Mitrovic and Stellan Ohlsson. Evaluation of a Constraint-Based Tutor for a Database Language. *International Journal of Artificial Intelligence in Education*, 10(3-4):238–256, 1999.

[99] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 373–376. ACM, 2004.

[100] Christian Murphy, Gail Kaiser, Kristin Loveland, and Sahar Hasan. Retina: Helping Students and Instructors Based on Observed Programming Activities. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 178–182. ACM, 2009.

[101] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. In *Working Group Reports from the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '02, pages 131–152. ACM, 2002.

[102] Cindy Norris, Frank Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree. ClockIt: Collecting Quantitative Data on How Beginning Software Developers Really Work. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 37–41. ACM, 2008.

[103] Andrei Papancea, Jaime Spacco, and David Hovemeyer. An Open Platform for Managing Short Programming Exercises. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 47–52. ACM, 2013.

[104] Nick Parlante. Nifty Reflections. *SIGCSE Bulletin*, 39(2):25–26, 2007.

[105] Dale Parsons and Patricia Haden. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education*, ACE '06, pages 157–163. Australian Computer Society, 2006.

[106] James H. Paterson, Ka Fai Cheng, and John Haddow. PatternCoder: A Programming Support Tool for Learning Binary Class Associations and Design Patterns. *Transactions on Computing Education*, 9(3):16:1–16:22, 2009.

[107] John Paxton. Live Programming as a Lecture Technique. *Journal of Computing Sciences in Colleges*, 18(2):51–56, 2002.

[108] Arnold Pears and Moritz Rogalli. mJeliot: A Tool for Enhanced Interactivity in Programming Instruction. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 16–22. ACM, 2011.

[109] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports from the 12th Annual SIGCSE Conference on Innovation*

and Technology in Computer Science Education, ITiCSE-WGR '07, pages 204–223. ACM, 2007.

[110] David N Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[111] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160. ACM, 2012.

[112] Wouter Poncin, Alexander Serebrenik, and Mark van den Brand. Mining Student Capstone Projects with FRASR and ProM. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*, SPLASH '11, pages 87–96. ACM, 2011.

[113] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[114] David Pritchard and Troy Vasiga. CS Circles: An In-browser Python Course for Beginners. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 591–596. ACM, 2013.

[115] Felix Raab, Christian Wolff, and Florian Echtler. RefactorPad: Editing Source Code on Touchscreens. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 223–228. ACM, 2013.

[116] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. VILLE: A Language-independent Program Visualization Tool. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research*, Koli Calling '07, pages 151–159. Australian Computer Society, 2007.

[117] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[118] Ma. Mercedes T. Rodrigo, Ryan S. Baker, Matthew C. Jadud, Anna Christine M. Amarra, Thomas Dy, Maria Beatriz V. Espejo-Lahoz, Sheryl Ann L. Lim, Sheila A.M.S. Pascua, Jessica O. Sugay, and Emily S. Tabanao. Affective and Behavioral Predictors of Novice Programmer Achievement. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, pages 156–160. ACM, 2009.

[119] Ma. Mercedes T. Rodrigo and Ryan S.J.d. Baker. Coarse-Grained Detection of Student Frustration in an Introductory Programming Course. In *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pages 75–80. ACM, 2009.

[120] Gruia-Catalin Roman and Kenneth C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26(12):11–24, 1993.

[121] Marc J. Rubin. The Effectiveness of Live-Coding to Teach Introductory Programming. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 651–656. ACM, 2013.

[122] Dean Sanders and Brian Dorn. Jeroo: A Tool for Introducing Object-oriented Programming. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 201–204. ACM, 2003.

[123] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An Introduction to Program Comprehension for Computer Science Educators. In *Working Group Reports from the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '10, pages 65–86. ACM, 2010.

[124] Otto Seppälä, Lauri Malmi, and Ari Korhonen. Observations on Student Misconceptions – A Case Study of the Build-Heap Algorithm. *Computer Science Education*, 16(3):241–255, 2006.

[125] Harsh Shah and Amruth N. Kumar. A Tutoring System for Parameter Passing in Programming Languages. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '02, pages 170–174. ACM, 2002.

[126] Beth Simon, Raymond Lister, and Sally Fincher. Multi-Institutional Computer Science Education Research: A Review of Recent Studies of Novice Understanding. In *Proceedings of the 36th Annual Frontiers in Education Conference*, FIE '06, pages 12–17. IEEE, 2006.

[127] Neeraj Singhal and Amruth N. Kumar. Facilitating Problem-Solving on Nested Selection Statements Using C/C++. In *Proceedings of the 30th Annual Frontiers in Education Conference*, FIE '00, pages T4C/3–T4C/6. IEEE, 2000.

[128] Teemu Sirkiä. A JavaScript Library for Visualizing Program Execution. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 189–190. ACM, 2013.

[129] Teemu Sirkiä and Juha Sorva. Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12, pages 19–28. ACM, 2012.

[130] John P. Smith III, Andrea A. Disessa, and Jeremy Roschelle. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The Journal of the Learning Sciences*, 3(2):115–163, 1994.

[131] Elliot Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[132] Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. Doctoral dissertation, Department of Computer Science and Engineering, Aalto University, 2012.

[133] Juha Sorva. Notional Machines and Introductory Programming Education. *Transactions on Computing Education*, 13(2):8:1–8:31, 2013.

[134] Juha Sorva, Ville Karavirta, and Lauri Malmi. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Transactions on Computing Education*, 13(4):15:1–15:64, 2013.

[135] Juha Sorva, Jan Lönnberg, and Lauri Malmi. Students' Ways of Experiencing Visual Program Simulation. *Computer Science Education*, 23(3):207–238, 2013.

[136] 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*, Koli Calling '10, pages 49–54. ACM, 2010.

[137] Juha Sorva and Teemu Sirkiä. Context-Sensitive Guidance in the UUhistle Program Visualization System. In *Proceedings of the Sixth Program Visualization Workshop*, PVW '11, pages 77–85. ACM, 2011.

[138] Jaime Spacco, Davide Fossati, John Stamper, and Kelly Rivers. Towards Improving Programming Habits to Create Better Computer Science Course Outcomes. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 243–248. ACM, 2013.

[139] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[140] James C. Spohrer and Elliot Soloway. Novice Mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, 1986.

[141] James G. Spohrer and Elliot Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers*, pages 230–251. Ablex Publishing, 1986.

[142] Margaret-Anne Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 181–191. IEEE, 2005.

[143] Edward R. Sykes and Franya Franek. A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java. In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education*, CATE '03, pages 78–83, 2003.

[144] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. Identifying At-Risk Novice Programmers through the Analysis of Online Protocols. In *Philippine Computing Society Congress*, 2008.

[145] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. Predicting At-Risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pages 85–92. ACM, 2011.

[146] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, Judith Bishop, Arjmand Samuel, and Tao Xie. The Future of Teaching

Programming is on Mobile Devices. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 156–161. ACM, 2012.

[147] Franklyn Turbak, Smaranda Sandu, Olivia Kotsopoulos, Emily Erdman, Erin Davis, and Karishma Chadha. Blocks Languages for Creating Tangible Artifacts. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '12, pages 137–144. IEEE, 2012.

[148] Ian Utting, Neil Brown, Michael Kölling, Davin McCall, and Philip Stevens. Web-Scale Data Gathering with BlueJ. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 1–4. ACM, 2012.

[149] Ernst Von Glasersfeld. A Constructivist Approach to Teaching. *Constructivism in Education*, 3:1–15, 1995.

[150] Tia Watts. The SFC Editor a Graphical Tool for Algorithm Development. *Journal of Computing Sciences in Colleges*, 20(2):73–85, 2004.

[151] Maximilian Rudolf Albrecht Wittmann, Matthew Bower, and Manolya Kavakli-Thorne. Using the SCORE Software Package to Analyse Novice Computer Graphics Programming. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 118–122. ACM, 2011.

[152] Denise Woit and David Mason. Effectiveness of Online Assessment. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 137–141. ACM, 2003.

[153] Wen-Hsiung Wu, Yen-Chun Jim Wu, Chun-Yu Chen, Hao-Yun Kao, Che-Hung Lin, and Sih-Han Huang. Review of Trends from Mobile Learning Studies: A Meta-Analysis. *Computers & Education*, 59(2):817–827, 2012.

Aalto-DD 69/2014

ISBN 978-952-60-5687-6
ISBN 978-952-60-5688-3 (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