Publication IV

Petri Ihantola, Ville Karavirta and Otto Seppälä. Automated visual feedback from programming assignments. In *Proceedings of the 6th Program Visualization Workshop (PVW '11)*, Technische Universität Darmstad, Darmstad, pages 87–95, 2011.

10 Automated Visual Feedback from Programming Assignments

Petri Ihantola, Ville Karavirta, Otto Seppälä
Aalto University
Department of Computer Science and Engineering
Finland

{petri.ihantola, ville.karavirta, otto.seppala}@aalto.fi

10.1 Introduction

Most modern assessment platforms for programming assignments are web-oriented and used with a browser (Douce et al., 2005). Students upload their work to the system and get (almost) immediate feedback from different aspects of their program, e.g. correctness, test coverage, style, etc (Ala-Mutka, 2005). We feel that the browser platform, however, is not used up to its full potential. Using HTML and JavaScript in the feedback itself could open new possibilities. In this paper, we explore ways for offering visual feedback in existing systems for automated assessment of programming assignments without radical changes to the assessment platforms. We also present initial results of effectiveness and some results from an attitude survey.

The use of visualizations in automated assessment of correctness of programming assignments is rare. Students get textual feedback from the functionality of their (textual) programs. Yet, as summarized by Helminen and Malmi (2010), visualizations are widely used in programming education, e.g. in programming learning environments such as Scratch, Jeliot, BlueJ and GreenFoot, and programming microworlds such as Karel.

The difficulty of creating visual feedback is probably the main reason of not using visualizations in automated assessment. The assessment of the student's solution is typically based on either unit tests or output comparison, both of which traditionally have a textual outcome - an assertion error message or a description of a difference between lines of text. Most assessment platforms also aren't very flexible in this sense and do not provide any easy or well documented way of going beyond unformatted text output.

When the textual output is used in a web environment we are however given a lot more options. Our key idea is: we can make feedback visual or even interactive by embedding HTML and JavaScript in the test descriptions of (JUnit) tests. This approach enables the use of visualizations in existing assessment systems without (or with minimal) modifications to the systems.

We explain the technical part of how to construct visual feedback in Section 2. In Section 3, we present initial results of an evaluation study done on a programming course comparing general visualizations with custom visualizations tailored for the domain of each exercise as well as textual feedback. Our findings and related research are discussed in Section 4 and Section 5 concludes our work.

10.2 Introducing Visual Feedback to Existing Assessment Platforms

Assume that we had a platform capable of providing feedback from student solutions that could include or even dynamically create visualizations that make the feedback more effective. For example, instead of using textual feedback to describe what is tested, we could illustrate that test case with visualizations. Complex data such as data structures constructed during the execution of programs could be shown as diagrams.

Our goal is to change the type of feedback delivered on an existing "text-based" platform. On high level, there are at least three different approaches to perform the change: *modifying the platform*, *writing a plugin* or *modifying the exercises/tests*.

All three approaches have their strengths and drawbacks. Modifying the platform requires deep knowledge of the platform architecture and implementation. Similarly, a familiarity with the plugin architecture and APIs is needed to implement the plugin approach. On the other hand, these two are the most flexible

approaches providing control over almost everything. Modifying the tests has the lowest learning curve, especially since the teacher typically writes the tests. On the downside, the approach is the most limited. Tests are executed in a sandbox and only the test results and test descriptions (e.g. names of tests and assert strings) are passed out from the box. Despite the challenges, we decided to explore what can be achieved by modifying only the tests.

10.2.1 Modifying Tests to Enable Visual Feedback

One approach to testing student solutions is based on unit testing. An exercise has multiple tests, and each test has multiple assertions that have to pass. When an assertion fails, a string describing that specific assertion is used as feedback. The fact that in an online environment the test results are shown inside a browser creates new possibilities not widely used. Feedback (e.g. assert strings) can contain HTML and JavaScript executed on client-side. For security and convenience to the teacher, web applications are expected to escape anything that could be misinterpreted as HTML. As embedding HTML is the very thing we want, the assessment platform should not perform escaping or should provide a way to selectively prevent it.

Drawing images manually is laborious and limited. For example, we can draw visualizations related to the model solution, but cannot anticipate and draw visualizations of all the possible data structure that the students' programs will produce. We wanted to explore how to construct such visualizations dynamically. There are at least two approaches to do this:

- HTML, CSS, and JavaScript can be used to draw the image entirely in the browser.
- Image tag with a dynamically constructed url pointing to an external web service (e.g. Google Chart API[1]) constructing the image can also be used.

We used both of these approaches – HTML and CSS to construct exercise specific visualizations and Google Chart API to draw generic object graphs. The following two subsections describe these.

Exercise Specific Visualizations

Exercise specific visualizations are, as the name suggests, written for a single exercise and they visualize the relevant parts of the exercise. We had exercises dealing with a video track editor and a chess game loader. In both cases we could visualize the state of the student's program and the expected state using a specialized visualization. For the chess exercise, we first tried a specialized JavaScript library to draw the board, but ended up doing it with HTML and CSS, as illustrated in Figure 10.1. In the video track exercise we wrote our own visualizations with HTML and CSS.

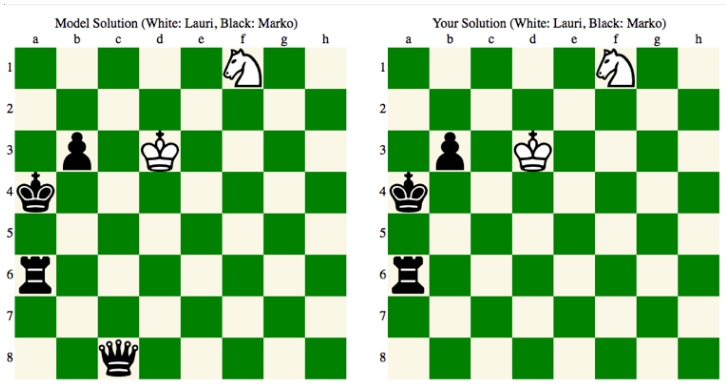

Figure 10.1: An example of exercise specific feedback from an assignment where students should construct an object graph based on a chess game state read from a text file.

[1] http://code.google.com/apis/chart/

The good thing about this approach is the ease of selecting only the relevant data from the assignment to be visualized and customizing presentations for the exercise. However, this does require some effort by the exercise designer implementing the visualizations in the tests. A more general way to give visual feedback is presented next.

Generic Visualizations based on Object Graphs

Automated assessment of a programming assignment is often based on comparing a student's solution against the model solution. This comparison can be performed by comparing textual outputs or object structures. In this work, we visualize the differences between object graphs.

Hamer (2004) has implemented a small tool called LJV to visualize Java object graphs. The tool uses Graphviz[2] as an intermediate language in the construction of visualizations. LJV constructs an object graph by traversing from the root node and passing the graph to a locally installed Graphviz. We wanted to push this idea further by removing the need of installing Graphviz. Unfortunately, we found the source code of LJV[3] only after we had implemented our own solution and were thus not able to build on that.

The Google Chart API is a RESTful web service capable of visualizing DOT-files (Graphviz's format). We can put a dot-description of a graph to the url of an image tag and get a rendered graph back from the service. To provide an example, `https://chart.googleapis.com/chart?cht=gv&chl=digraph{A->B->C->A}` results in an image of a cyclic graph with three nodes. We can pass such feedback as HTML and students' browsers will call the Chart API to render the feedback.

In our early experiments we found it hard to compare two large graphs. To overcome this problem, we limited information and provided visual hints to locate the differences between the graphs.

Not all parts of object graphs are relevant to the assessment. For example, in an assignment where students are asked to build a binary search tree, only the references pointing to the left child, the right child, and the data are relevant. If a student stores some extra information to a node (e.g. for debugging purposes), this information is not relevant. Thus, a way to configure how object graphs are extracted is needed. LJV has a configuration interface to do similar tasks. In our prototype, we offer something similar. Teachers can control the construction of graphs by implementing one method for each object type. It is also possible to control if selected objects (e.g. Strings) should not be visualized as independent objects (i.e. boxes) but as fake primitives inside the object referring to them. We found this necessary in order to keep our images readable.

Visualizing Differences Between Object Graphs

Before visualizing differences between object graphs we need to know how the two graphs differ, or alternatively, which parts of the graphs match. We decided to draw both graphs and use colors to highlight differences between nodes and references of the graphs – green to mark that an element has a matching pair (it is correct), red to mark that there is something wrong, and black to mark that those parts of a graph have not been compared because of earlier problems.

Comparison starts with root nodes from both graphs and proceeds in breadth first manner to search for unmatching nodes. Visualizations to highlight the differences between the two graphs are constructed during the comparison algorithm. Nodes in both graphs are compared based on:

1. **The data stored inside the nodes**. If the data are equal, both nodes are colored green and otherwise nodes are colored red and the comparison stops.

2. **The lists of references going out from the nodes**. Labels of these references can be empty in which case only the length of the lists is important or the labels can have semantic meaning, e.g. names of the references (e.g. right, left, data). Comparison of lists is based on the difference of the lists of labels of outgoing edges. Difference is calculated with the java-diff-utils library[4]. Comparing two lists, A and B, produces a patch describing how to modify B to make it equal with A. A patch is a list of deltas where each delta is either insert, edit, or delete. We calculate two patches – one from the model solution to the student's solution (P1) and one from the student's solution to the model solution (P2). Inserts in the patch from A to B are deletes in the patch from B to A and vice versa. To visualize a patch (i.e. a difference in outgoing edges of a node), we highlight deletes

[2] `http://www.graphviz.org/`
[3] `http://www.cs.auckland.ac.nz/~j-hamer/LJV.java`
[4] `http://code.google.com/p/java-diff-utils/`

and edits of P1 in the graph of the model solution and deletes and edits of P2 in the graph of student's solution. The model answer can also contain references colored in red. This means that those references are different in or missing from the student's solution. An example from such case is provided in Figure 10.2.

The comparison of lists of references going out from the nodes can be illustrated with the following two examples. First, A = [1, 2, 3, 4, 5] and B = [1, 2 ,5] is a simple example leading to highlighting 3 and 4 from A. Those need to be deleted from A to get B. A case where patches P1 and P2 both have deletes is more complicated. For example, A = [1, 2, 3, 4] and B = [1, 2 ,5] results in highlighting 3 and 4 in A and 5 in B.

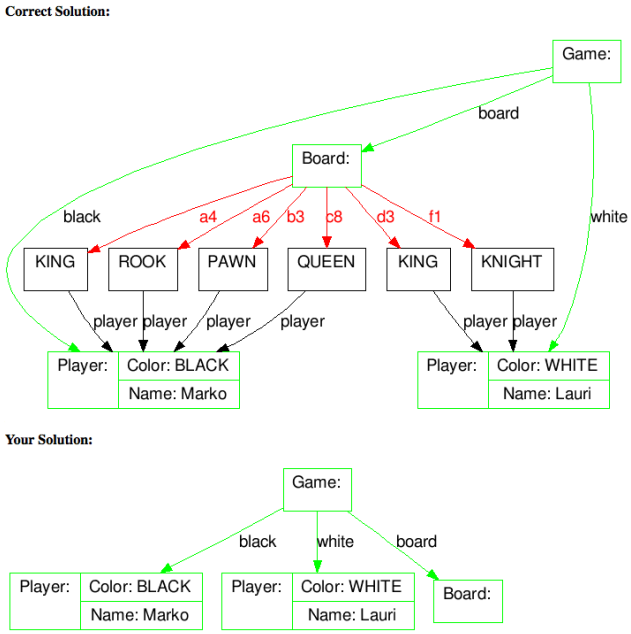

Figure 10.2: An example of generic feedback related to chess board construction exercise.

Finally, layouts of the graphs need to be similar to maintain the readability. Chart API handles the layout and if the two graphs are not identical, matching nodes can be in different positions. Based on our experiment, it is often enough if the order of references from a common node is controlled. This can be controlled with a single setting in the graph description.

10.3 Evaluation Study

10.3.1 Research Setup

We had our research setup on Intermediate Programming in Java course at Aalto University in Spring 2011. The course teaches object oriented programming in Java and is worth 6 ECTS-credits. The course has a 5 ECTS-credits CS1 course as a prerequisite. Automatic assessment is used on both courses. The exercises count for 30 percent of the course grade. All exercises were assessed with Web-CAT (Edwards, 2003). The number of resubmissions was not limited. Support groups where students could come and ask for help were provided but the attendance was not mandatory.

Students participating in the experiment were divided randomly into three groups – Groups A, B and C. First exercise round was designed to be the pretest ensuring that the groups are not different. Therefore, all groups got textual feedback from this round. In the second round, group A got textual feedback, B received specialized visualizations, and C got object graph based generic feedback. In the

Table 10.1: Feedback versions of the groups on different assignments on the course. Numbers in parentheses are the number of students who got visual feedback from the number of students who submitted a solution.

Assignment	Group A	Group B	Group C
1.1	textual (71)	textual (63)	textual (66)
1.2	textual (67)	textual (61)	textual (63)
1.3	textual (62)	textual (54)	textual (56)
2.2	textual (65)	specific visualization (9 / 57)	generic visualization (16 / 61)
2.3	textual (49)	specific visualization (32 / 44)	generic visualization (43 / 55)
3.1	generic visualization (21 / 48)	specific visualization (18 / 53)	textual (57)
3.2	generic visualization (21 / 43)	specific visualization (14 / 43)	textual (51)

third and last round, feedback types between groups A and C were swapped. The rationale behind this setup is that we were most interested to compare textual feedback against our generic, object graph based visualizations. Grading criterion between the groups were equal.

We gave visual feedback based on the comparison of object graphs. This implies that students had to be able to construct something comparable to see visualizations. If a student had a bug in his or her program that caused an exception, visual feedback was not possible. In this case, the feedback was the same as for the group getting textual feedback.

To compare the groups, we collected submission times and the grades/feedback from Web-CAT. After all the exercises were closed, we asked students to take a voluntary questionnaire about their attitudes and opinions related to visual feedback.

10.3.2 Preliminary Results

Submission Data

The number of students from each group who submitted something to the exercises is summarized in Table 10.1. In exercises where a group got visual feedback, two numbers are given. The first number is how many students got visualizations and the second the total number of students. From this data, exercise 2.3 is the most interesting. From the other assignments, majority of the students got traditional feedback making the comparison between groups difficult.

For each student and each exercise we first identified the best submission. This is the submission where the product of the three factors of grading (i.e. percentage of teacher's test passing, test coverage of student's own tests, and the percentage of student's own tests passing) is the highest. From this submission we retrieved the factors of the grade.

For each exercise and for each of the three factors, neither Kruskal-Wallis nor one way ANOVA were able to reject the null hypothesis that the groups are similar ($p>0.05$).

Although no differences were found, details of distributions in assignments with visual feedback are of interest. In all exercises and with every feedback type, only few outliers did not get full 100% from their own tests. Boxplots describing the distributions of teacher's tests passing and coverage of students' own tests are in Figure 10.3. What we can interpret from the data is that exercises 2.3 and 3.2 were the most difficult. The fact that students often write bad tests from where they get good scores has also been reported by Aaltonen et al. (2010).

As most students got full points from many of the assignments, it is more interesting to analyze how they got to this situation. Thus, for each student and each assignment, the total number of submissions was calculated. Again, no statistically significant differences were found between the groups ($p>0.05$).

Questionnaire

An interesting question is whether they would have liked to get more textual or more visual feedback. Results of this question are shown in Table 10.2 divided between students who got visual feedback and who did not. It can clearly be seen[5] that those who got visual feedback are more positive towards it, whereas students who only got textual feedback would prefer it in the future as well. How much did the students use different types of feedback was also triangulated with multiple questions. The answer is that most students use all the time all the feedback available.

[5] In fact, the difference between the groups is statistically significant (Kruskal-Wallis test, $p<0.05$).

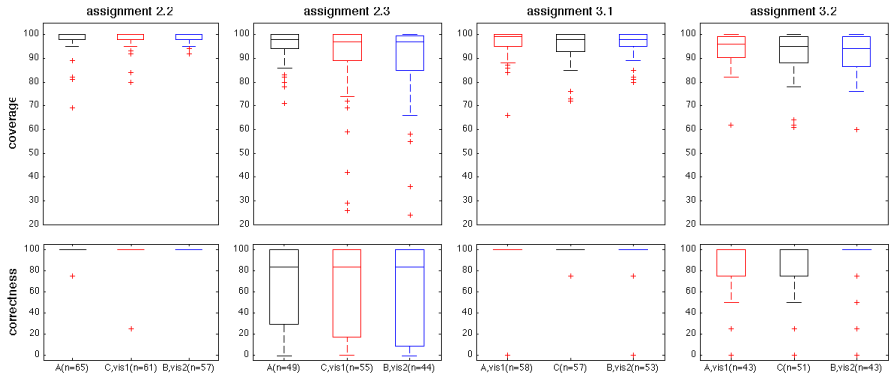

Figure 10.3: Boxplots related the performance metrics of different groups in the exercises where some students got visual feedback. Groups labeled with vis1 and vis2 got generic graph visualizations and exercise specific visualizations, respectively.

Free comments about visual feedback included positive things like: more efficient to deduce where the problem is in the code, more exact, more compact, and _always_ nicer to look at. Negative things mentioned were: the visualization was a mess, visual feedback was confusing but helpful once one understood what it meant, and visual feedback should include more verbal description how it should be interpreted. Two students commented that verbal feedback is in most cases more useful, unless the exercise deals with a problem naturally visualized (like the chessboard exercise). Furthermore, one comment mentioned verbal feedback to be more clear while another noted that verbal feedback requires knowledge about the terminology and understanding of the logic of the creator of the exercise.

10.4 Discussion and Related Research

Ma et al. (2009) have observed students to perform better with visualizations designed specifically to visualize object references are used when compared to generic visualizations of Jeliot 3. Our data does not provide statistically significant support to this. Partially this can be because in many exercises, students solved assignments without seeing visualizations. This means that after their programs managed to run without exceptions, the data structures constructed were also correct. Exercise 2.3 was the most difficult and also the only one where the majority of the students needed/got visual feedback. In this exercise, students getting textual feedback performed the best, graph based being the second and specialized feedback third (see Figure 10.3). Differences, however, were not significant.

The course we used in our study has used almost the same assignments for several years. Textual feedback from these exercises has also matured. In order to develop textual feedback of the same quality, a substantial amount of work is needed. Accordingly, designing good custom visualization can also take time. Thus, we argue that constructing graph based visual feedback is the most effortless. Therefore, visual feedback not performing significantly worse than mature textual feedback is a good result.

Visualizations seem to be useful only if assignments are difficult enough or with certain types of assignments. Programs should not fail before they return something to be compared and that something should difficult enough to construct. To better understand when to use visual feedback, different types of errors in different types of programming assignments needs to be studied more.

Table 10.2: Attitude towards visual or textual feedback. Students are divided between those who got visual feedback and those who did not.

Visual Feedback	Only Visual	Mostly Visual	Mostly Textual	Only Textual
Yes	0 (0%)	28 (67%)	13 (31%)	1 (2%)
No	1 (5%)	6 (32%)	10 (53%)	2 (11%)

Ben-Bassat Levy et al. (2003) have pointed out that visualizations can help by providing a common vocabulary and model to better understand program execution. Verbal feedback in our survey data supports this. Despite the benefits of visualization, Ben-Bassat Levy et al. conclude that *"the interpretation of the animation itself is non-trivial and must be explicitly taught."* Based on the feedback we got from our students, the same applies for static visualizations as students requested feedback images to be better explained.

The way visualizations are used is often more important than what the visualizations contain (Hundhausen et al., 2002). Introducing visualizations through HTML/JavaScript has the potential of making feedback interactive. For example, feedback could contain interactive algorithm animations or questions directing the construction of visual feedback. In future, we hope to see JavaScript libraries or web services supporting construction of these kinds of feedback. Our use of the Chart API to draw object graphs is a start in the right direction.

Security of assessment platforms needs to be improved in the future (Ihantola et al., 2010). This is emphasized when the system allows educators more freedom over the feedback creation process. Reducing responsibilities of the assessment server by externalizing such execution should help securing the server. While teachers should be allowed more control over the feedback markup, assessment systems should treat data from students' programs with care. In many assessment platforms, submissions are viewed online. Including JavaScript in the code of a submission is a possible attack vector. In addition, teachers should be aware what data they pass to external services.

10.5 Conclusions

Embedding HTML into feedback provides a flexible approach to introduce more visual feedback to current automated assessment systems used to grade programming assignments. Benefits of our approach, compared to applets or standalone visualization software, are that the approach is lightweight, it can be used with assessment tools already in place, and assignments designed with such feedback are relatively easy to port from one assessment platform to another – assuming both platforms are used with a browser.

For authors of assessment systems, supporting this is straightforward. The feedback should not be HTML escaped or there should be an option to selectively prevent escaping.

For the teacher creating the test, generic object graph visualization can give a more effortless way compared to exercise specific visualizations or textual feedback. The preliminary results of our evaluation of such feedback showed no (statistically significant) differences in student's results between any of the feedback types.

Bibliography

Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *OOPSLA companion*, SPLASH '10, pages 153–160. ACM, 2010.

Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):1–13, 2005. ISSN 1531-4278. doi: http://doi.acm.org/10.1145/1163405.1163409.

Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA Companion*, pages 148–155. ACM, 2003. ISBN 1-58113-751-6. doi: http://doi.acm.org/10.1145/949344.949390.

John Hamer. A lightweight visualizer for java. In *Proceedings of Third Program Visualization Workshop*, pages 54–61, 2004.

Juha Helminen and Lauri Malmi. Jype - a program visualization and programming exercise tool for python. In *Proceedings of SOFTVIS '10*, pages 153–162. ACM, 2010.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *JVLC*, 13(3):259–290, June 2002.

Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of Koli Calling '10*, pages 86–93. ACM, 2010.

Linxiao Ma, John D. Ferguson, Marc Roper, Isla Ross, and Murray Wood. Using cognitive conflict and Jeliot visualisations to improve mental models. In *Proceedings of ITiCSE '09:*, 2009.