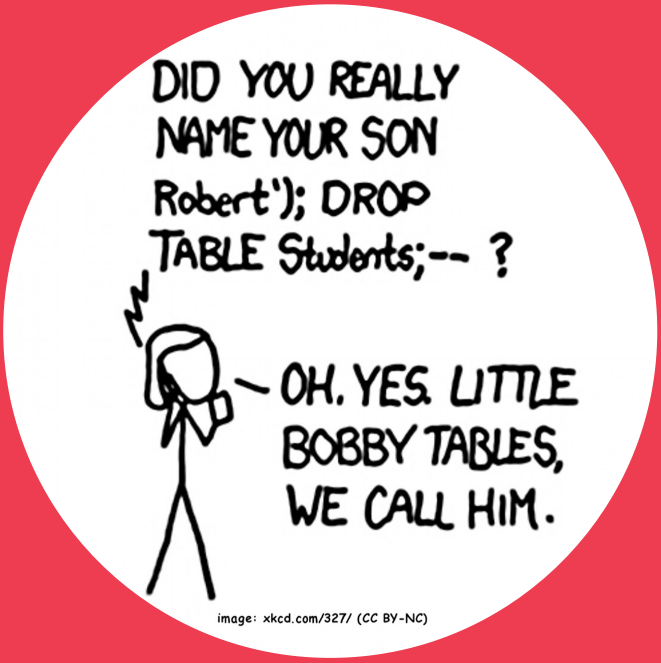
Department of Computer Science and Engineering

Automated Assessment of Programming Assignments

Visual Feedback, Assignment Mobility, and Assessment of Students' Testing Skills

Petri Ihantola


DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
image: xkcd.com/327/ (CC BY-NC)

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 131/2011

Automated Assessment of Programming Assignments: Visual Feedback, Assignment Mobility, and Assessment of Students' Testing Skills

Petri Ihantola

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 in T-building at the Aalto University School of Science (Espoo, Finland) on the 9th of December 2011 at 12 noon.

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

Supervisor
Professor Lauri Malmi

Instructors
D.Sc. (Tech) Ari Korhonen
D.Sc. (Tech) Ville Karavirta

Preliminary examiners
Associate Professor Stephen H. Edwards, Virginia Tech, USA
Associate Professor Mike Joy, University of Warwick, UK.

Opponent
Professor Tapio Salakoski, University of Turku, Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 131/2011

© Petri Ihantola

ISBN 978-952-60-4397-5 (printed)
ISBN 978-952-60-4398-2 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Unigrafia Oy
Helsinki 2011

Finland

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

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

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

Abstract

Author
Petri Ihantola

Name of the doctoral dissertation
Automated Assessment of Programming Assignments: Visual Feedback, Assignment Mobility, and Assessment of Students' Testing Skills

Publisher Aalto University School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 131/2011

Field of research Software Systems

Manuscript submitted 12 July 2011 **Manuscript revised** 24 October 2011

Date of the defence 9 December 2011 **Language** English

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

Abstract

The main objective of this thesis is to improve the automated assessment of programming assignments from the perspective of assessment tool developers.

We have developed visual feedback on functionality of students' programs and explored methods to control the level of detail in visual feedback. We have found that visual feedback does not require major changes to existing assessment platforms. Most modern platforms are web based, creating an opportunity to describe visualizations in JavaScript and HTML embedded into textual feedback. Our preliminary results on the effectiveness of automatic visual feedback indicate that students perform equally well with visual and textual feedback. However, visual feedback based on automatically extracted object graphs can take less time to prepare than textual feedback of good quality.

We have also developed programming assignments that are easier to port from one server environment to another by performing assessment on the client-side. This not only makes it easier to use the same assignments in different server environments but also removes the need for sandboxing the execution of students' programs. The approach will likely become more important in the future together with interactive study materials becoming more popular. Client-side assessment is more suitable for self-studying material than for grading because assessment results sent by a client are often too easy to falsify.

Testing is an important part of programming and automated assessment should also cover students' self-written tests. We have analyzed how students behave when they are rewarded for structural test coverage (e.g. line coverage) and found that this can lead students to write tests with good coverage but with poor ability to detect faulty programs. Mutation analysis, where a large number of (faulty) programs are automatically derived from the program under test, turns out to be an effective way to detect tests otherwise fooling our assessment systems. Applying mutation analysis directly for grading is problematic because some of the derived programs are equivalent with the original and some assignments or solution strategies generate more equivalent mutants than others.

Keywords teaching programming, automated assessment, visual feedback, software visualization, portability, mobility, testing, mutation analysis

ISBN (printed) 978-952-60-4397-5 **ISBN (pdf)** 978-952-60-4398-2

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

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 228 **The dissertation can be read at** http://lib.tkk.fi/Diss/

Aalto-yliopisto

Tiivistelmä

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

Tekijä Petri Ihantola	
Väitöskirjan nimi Ohjelmointitehtävien automaattinen arviointi: visuaalinen palaute, tehtävien siirrettävyys ja testaustaitojen arviointi	
Julkaisija Aalto-yliopiston perustieteiden korkeakoulu	
Yksikkö Tietotekniikan laitos	
Sarja Aalto University publication series DOCTORAL DISSERTATIONS 131/2011	
Tutkimusala Ohjelmistojärjestelmät	
Käsikirjoituksen pvm 12.07.2011	**Korjatun käsikirjoituksen pvm** 24.10.2011
Väitöspäivä 09.12.2011	**Kieli** Englanti
☐ **Monografia**	☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Väitöskirjassa tarkastellaan ohjelmointitehtävien automaattiseen arviointiin soveltuvien työkalujen kehittämistä lähinnä ohjelmoijan näkökulmasta.

Väitöskirjassa kehitetään ohjelmien virheellistä toimintaa havainnollistavaa visuaalista palautetta ja visuaalisaatioiden yksityiskohtaisuuden hallintaa. Visuaalinen palaute ei välttämättä vaadi suuria muutoksia olemassa oleviin arviointijärjestemiin. Useimmat järjestelmistä ovat selainpohjaisia, jolloin visuaalinen palaute on mahdollista toteuttaa sisällyttämällä tekstuaaliseen palautteeseen JavaScript- ja HTML-osuuksia. Alustavat tulokset visuaalisen palautteen hyödyllisyydestä viittaavat siihen, että opiskelijat suoriutuvat yhtä hyvin riippumatta siitä saivatko he visuaalista vai sanallista palautetta. Automaattinen, oliograafeja visualisoiva palaute on kuitenkin joissakin tilanteissa vaivattomampaa laatia, kuin hyvä sanallinen palaute.

Väitöskirjan toinen tutkimusalue on ohjelmointitehtävien siirrettävyys palvelinympäristöjen välillä. Tarkastelluista lähestymistavoista mielenkiintoisimmaksi osoittautui ohjelmien tarkastaminen opiskelijan selaimessa. Lähestymistavan lisäetuna on tuntemattoman koodin suorittaminen ainoastaan palvelimen ulkopuolella. Selaimessa itsensä tarkastavien ohjelmointitehtävien merkitys tullee kasvamaan vuorovaikutteisten oppimateriaalien yleistyessä. Selainpohjainen tarkastaminen soveltuu heikosti arvosteluun mutta hyvin itseopiskelumateriaaleihin, koska selaimen palvelimelle lähettämien tulosten väärentäminen on usein liian helppoa.

Testaus on osa ohjelmointia, ja automaattista palautetta tulee antaa myös opiskelijoiden omista testiohjelmista. Nykyisin opiskelijoiden testeistään saama palaute perustuu usein rakenteelliseen kattavuuteen (esim. rivikattavuus), jolloin osa opiskelijoista kirjoittaa testejä, jotka erinomaisesta testikattavuudestaan huolimatta eivät kykene erottamaan virheellistä ohjelmaa oikeasta. Väitöskirjan viimeinen kontribuutio onkin osoittaa, että mutaatiotestaus, jossa testattavasta ohjelmasta johdetaan suuri joukko (virheellisiä) ohjelmia, on tehokas keino havaita testejä, jotka aikaisemmin saattoivat huiputtaa automaattista arviointia. Arvosanan muodostaminen mutaatiotestauksen perusteella on kuitenkin ongelmallista, sillä osa automaattisesti johdetuista ohjelmista ei välttämättä eroa riittävästi alkuperäisestä ollakseen virheellisiä.

Avainsanat ohjelmoinnin opetus, automaattinen arviointi, visualinen palaute, ohjelmien havainnollistaminen, siirrettävyys, testaus, mutataatiotestaus		
ISBN (painettu) 978-952-60-4397-5	**ISBN (pdf)** 978-952-60-4398-2	
ISSN-L 1799-4934	**ISSN (painettu)** 1799-4934	**ISSN (pdf)** 1799-4942
Julkaisupaikka Espoo	**Painopaikka** Helsinki	**Vuosi** 2011
Sivumäärä 228	**Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/	

Rakkaalle vaimolleni Maijalle
ja lapsillemme.

Preface

In the beginning of my undergraduate studies, over ten years ago, I did not know what I was about to start. After my freshman year at Helsinki University of Technology, I got recruited to the TRAKLA2 research group and started work as a TA in programming courses. Soon, I was doing more teaching and maintaining some of our assessment tools. Finally, I drifted into running a programming course with a dozen TAs and hundreds of students.

In 2007, I left teaching, research, and Finland for two years to work for Google. I moved first to Ireland and then to Switzerland. I gained new skills, friends, and experiences. I believe that this adventure speeded up my research rather than slowing it down, and enabled me to finish this manuscript, indeed, earlier.

I have received many great opportunities, for which I am truly grateful. I have had the pleasure to see the birth and growth of computing education research in our department. Some of my publications are from the very early days of this development while others are from this year. I have met many wonderful personalities who have made this manuscript possible – too many that I could list you all. Thank you!

pre-examination, prof. Stephen H. Edwards, and prof. Mike Joy, and

Juha Sorva, Otto Seppälä, pointed out so many relevant references,

supervision, instructions, guidance, Prof. Lauri Malmi, Dr.Sc. (Tech) Ari Korhonen, Dr.Sc. (Tech) Ville Karavirta,

co-authors of the publications, Ari Korhonen, Lauri Malmi, Jussi Nikander, Otto Seppälä, Tuukka Ahoniemi, Ville Karavirta, Kalle Aaltonen, Atanas Radenski, M. Gloria Sánchez-Torrubia, Guido Rößling, Myles McNally, Pierluigi Crescenzi,

advice on language, Otto Seppälä, Juha Helminen, Juha Sorva, Pietari Gröhn,

Juho Komu, Eila Ranta, Raimo Ruottinen, janitors who patiently kept letting me back in my room after I forgot my keys there,

other help, child care and Toini & Juhani Tenhunen, Sirpa & Risto Ihantola,

Learning + Technology Group, Lauri Malmi, Ari Korhonen, Ville Karavirta, Jan Lönnberg, Juha Sorva, Tapio Auvinen, Lasse Hakulinen, Juha Helminen, Otto Seppälä, Ahmad Taherkhani, Teemu Sirkiä, Teemu Koskinen, ...,

Kimmo Roimela, Karri Huhtanen, friends, roomies who studied CS when I started with physics,

Hannu Peltola cheered me up with not too easy jokes,

friends from Google, Damian Gajda, Marcin Brodziak, Greg Bullock, Jyrki Alakuijala, Toni Timonen, Hannu Helminen, Tuomo Kokko, Markus Clermont, J. Pablo Fernández, Liam Bedford, Yaroslav Samchuk, Robert Nilson, Albert Chen, Mehmet Akın, Brian Quinlan, Dragos Dociu, Daniel Aioanei, Jean-Christophe Lilot, Michal Kozak,

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 Ari Korhonen, Lauri Malmi, Jussi Nikander, and Petri Ihantola (ne Tenhunen). Interaction and feedback in automatically assessed algorithm simulation exercises. *Journal of Information Technology Education*, pages 241–255, vol 2, 2002.

II 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)*, ACM, New York, NY, USA, pages 86–93, 2010.

III Petri Ihantola. Creating and visualizing test data from programming exercises. *Informatics in education 6(1)*, pages 81–102, January 2007.

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.

V Petri Ihantola, Ville Karavirta, Ari Korhonen, and Jussi Nikander. Taxonomy of effortless creation of algorithm visualizations. *Proceedings of the 2005 International workshop on Computing Education Research (ICER '05)*, ACM, New York, NY, USA, pages 123–133, 2005.

VI Guido Rößling, Myles McNally, Pierluigi Crescenzi, Atanas Radenski, Petri Ihantola, and M. Gloria Sánchez-Torrubia. Adapting moodle to better support CS education. In *Proceedings of the 2010 ITiCSE working group reports (ITiCSE-WGR '10)*, ACM, New York, NY, USA, pages 15–27, 2010.

VII Petri Ihantola and Ville Karavirta. Two-dimensional Parson's puzzles: the concept, tools and first observations. *Journal of Information Technology Education and Innovations in Practice*, vol. 10, pages 119–132, 2011.

VIII Ville Karavirta and Petri Ihantola. Automatic assessment of JavaScript exercises. In *roceedings of 1st Educators' Day on Web Engineering Curricula (WECU 2010)*, Volume 607 of *CEUR-WS*, Vienna, Austria, pages P9:1–P9:10, 2010.

IX Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (SPLASH '10)*, ACM, New York, NY, USA, 153–160, 2010.

Author's Contribution

Publication I describes how learning styles can be applied to design assignments and automatic feedback. Although we mainly discuss algorithmic assignments, we also show how learning styles can inspire the construction of automatically assessed programming assignments.

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

Publication II presents a systematic literature review of recent (2006–2010) development in the field of automated assessment tools for grading programming exercises.

I am the corresponding author of this paper and in charge of combining the survey results. The actual survey, however, was equally divided among all the authors.

Publication III explains how generalized symbolic execution with lazy initialization can be used to visualize automatically generated test data on a high level of abstraction.

I am the sole author of this work.

Publication IV describes how HTML and JavaScript enable visual feedback with minimal or no modifications to the existing assessment platforms. A small evaluation study of the effectiveness of visual feedback is also presented.

I am the corresponding author with the original idea and in charge of the research. I analyzed the data and wrote/implemented the parts related to the object graph based feedback.

Publication V introduces a taxonomy of effortless creation of algorithm visualizations. Whereas Publication I discusses different learning styles broadly, this paper focuses on the difficulty of supporting visual learners.

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

Publication VI is the outcome of one of the international ITiCSE 2010 working groups. The paper describes how Moodle, a popular learning management system, supports teaching CS topics and what features are missed most by the teachers.

I am mainly responsible for the part of our literature survey that focused on automated assessment of programming assignments in Moodle. In addition, I made significant contributions to designing the questionnaire sent to teachers and analyzing the answers together with Pierluigi Crescenzi and actively participated in designing, structuring and writing the final report.

Publication VIII surveys professional development tools that could be useful in building a system that gives automatic feedback on JavaScript programming assignments. We further explain how we built simple automatically assessed JavaScript assignments as a proof-of-concept implementation. One important feature of our platform is that all the assessment is performed inside students' browsers instead of a server.

Both authors contributed evenly to all parts of this paper and to the development of the tools.

Publication VII explains how the concept of Parson's puzzles, a type of programming exercise, can be extended. We also implemented a system where students can solve these exercises and where teachers can create the puzzles. We report first observations of how experienced programmers solve these puzzles.

I was the corresponding author with the original idea but both authors contributed evenly to all parts of this paper and development of the tools.

Publication IX demonstrates how feedback based on structural coverage of students' tests can actually tell very little about the ability of the tests to find defects. We also discuss the use of mutation testing to complement assessment based on structural coverage.

I contributed significantly to all parts of this paper and came up with the idea. Kalle Aaltonen analyzed the data in his Master's thesis, which I instructed. After Kalle's thesis, we (the authors of Publication IX) wrote Publication IX based on Kalle's thesis.

Contents

1. Introduction

Programming skills are needed not only by computer scientists, but also in other disciplines. Many students do take programming courses, especially on beginner and intermediate levels. In all courses, large or small, assessment and prompt feedback are important aspects of learning.

Assessment provides feedback for the learner and the teacher about the learning process – from the level of a whole course down to a single student and some specific topic being assessed. Continuous assessment during a programming course ensures that the students practice and that they get feedback on the quality of their solutions. Even for small class sizes, providing quality assessment manually means that feedback cannot be as immediate as in one-to-one tutoring. As the class size grows, the amount of assessed work has to be cut down or rationalized in some other way. Automated assessment, however, allows instant feedback regardless of the class size.

1.1 Thesis Scope

Learners and *teachers* interact with the assessment tools that are the scope of my thesis. My point of view is technical: I explore technologies potentially suitable for building better assessment tools. This technical perspective differs from that of educational science. In particular, how learning occurs and details of how automated assessment impacts learning are out of my scope.

Learning to program can be supported with different exercises and many of those can be assessed automatically. Carter et al. [19] lists multiple-choice questions, questions with textual answers (e.g. essays), assignments with visual answers (e.g. diagrams), peer assessed assignments and programming assignments – all being assessed automatically. In this thesis, I focus on automated assessment of *programming assignments* where students get feedback from the correctness of their programs and the quality of their tests. I have ruled out of scope, for example, feedback from style and plagiarism detection. From now on, assessment will refer to assessment of programming assignments.

Level of automation in the assessment process varies from managing manual feedback down to automating the whole process [4, 19]. In this thesis, I research fully *automated assessment*[1] of programming assignments.

1.2 Automated Assessment and Feedback

Feedback is the outcome of assessment. It can be further divided between *formative feedback* that aims to improve learning, and *summative feedback* that is a judgment. Teachers use summative feedback for grading purposes. Learners, on the other hand, are interested in both – summative feedback as it tells them the grade and formative feedback as it tells how to improve.

Automated assessment takes place in an *assessment platform* (e.g. Web-CAT [30] or BOSS [51, 50]). Most modern assessment platforms are web based [28]. Students use them with their browsers and submit programs to the server where the programs are typically tested. Many of the platforms include course management features – taking care of submissions, finding the best submission and collecting results for the teacher.

Figure 1.1 gives an overview of automated assessment with the assessment platform at the center. Teachers and learners interact through the platform. Ideally, automatically assessed assignments, just like all other assignments, are designed based on learning objectives. What makes automatically assessed assignments different is the strong influence of the selected assessment platform. Each platform has its own strengths and weaknesses that affect the assignment design. To highlight that an automatically assessed assignment is tied to the assessment platform, I separate *assignment* from *assignment implementation*. An assignment is a task for students to perform together with an idea for how to assess that performance. An assignment implementation is how an assignment is implemented on a specific assessment platform.

Students interact with the platform by uploading their programs to be assessed. The feedback is typically almost immediate and it can be deliv-

[1]I also use *automatic assessment* as a synonym because both terms are used in the original articles.

ered both to teachers and learners, as also illustrated in Figure 1.1. Later, students can revise their submissions, fix them and submit again to get new feedback.

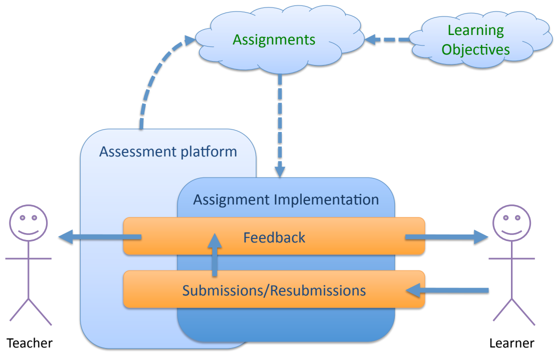


Figure 1.1. Teacher, learner and an automatically assessed assignment. Solid arrows denote data flows – learner submitting his or her program and getting feedback, for example. Dashed arrows illustrate influence of something on something else – assignments, for example, are influenced by the learning objectives and the limitations/strengths of the assessment platform.

1.3 Overarching Research Questions

The goal of this thesis is to improve automated assessment of programming assignments by 1) adding visual elements to formative feedback, 2) implementing assignments that are easier to port from one assessment platform to another and 3) exploring new ways to give feedback on students' testing skills. These three themes are the titles of the following subsections. Short motivation and high-level research questions, related to each of these themes, are presented in this section. These questions are operationalized later in Chapters 4, 5, and 6 – after the background presented in Chapters 2 and 3.

1.3.1 Visual Feedback

The use of visualizations in automated assessment of programming assignments is rare. Students get textual feedback on various aspects[2] of their textual programs. Yet, visualizations are widely applied elsewhere

[2]These will be discussed in Section 3.1.2.

in computer science (CS) education to support the learning process (see Chapter 2). Visual feedback on functionality of students' programs is especially rare. The gap in between the use of visualization and automated assessment of programming assignments motivates the following research question:

How to provide visual feedback on automatically assessed programming assignments?

I am interested in developing visualizations describing behaviors and functionalities of students' programs. Figure 1.2 illustrates what this can look like. The technical side of how visualizations can be constructed and included in the existing assessment platforms will be presented in Chapter 4. In addition, usefulness of visual feedback will also be discussed in Chapter 4.

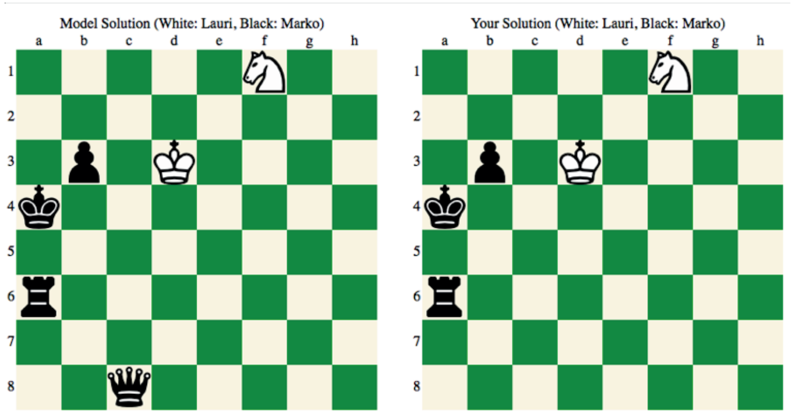

Model Solution (White: Lauri, Black: Marko)
Your Solution (White: Lauri, Black: Marko)
a
b
c
d
e
f
g
h
1
2
3
4
5
6
7
8

Figure 1.2. Visual feedback from an assignment where the task is to read a text file and initialize chess board data structures based on the data. The images are shown to a student after he or she makes a submission to the assignment. The students can compare the images to find how his or her output differs from the model output. (figure originally published in Publication IV).

1.3.2 Assignment Mobility

An assignment implementation is typically tightly coupled with the assessment platform. A teacher implements assignments for a specific assessment platform making it difficult to reuse the assignments on other platforms. For example, how the assessment process is carried out or how different factors affecting the grade are weighted are defined differently on most assessment platforms. My second research question is:

How to allow assignment implementations and visual feedback to be more independent from assessment platforms?

I use the term *assignment mobility* throughout my thesis in the contexts of sharing assignment implementations between teachers and porting assignments from one assessment platform to another.

A common format that multiple assessment platforms could use to share assignments, as proposed by Edwards et al. [31], would be the obvious solution to the problem. Unfortunately, so far no such format has become popular. I chose instead to explore the potential of three approaches to support assignment mobility in Chapter 5:

1. Integrating visualization tools into the assessment platforms.
2. Integrating assessment into general purpose learning management systems already used by many.
3. Moving assessment and other responsibilities from the server to the clients (i.e. browsers).

1.3.3 Assessing Testing Skills

For several years, students in a programming course at my university have been required to test their own programs and submit their tests for assessment. The assessment has been based on structural test coverage and static analysis to ensure that tests indeed have meaningful assertions. Although students score well, we felt that at least some of the students get good scores from tests of really poor quality. Thus, my third and the last top level research question is

How to better evaluate students' testing skills?

The question will be operationalized in Chapter 6 where an alternative test quality metric (mutation analysis [24]) will be compared against the previously used metric.

1.4 Thesis Structure

This summary part (Chapters 1 – 7) of my article dissertation highlights how publications I – IX are related to the research themes explained in the previous section. The rest of this thesis is organized in the following manner:

Chapter 2 discusses some educational theories and gives background to the context where the results of this work can be applied.

Chapter 3 summarizes the state of the art in automated assessment of programming assignments.

Chapters 4, 5 and 6 focus on each of the research topics – *visual feedback*, *mobility of assignments* and *assessing testing skills*, respectively.

Chapter 7 concludes this thesis with a summary of what has been done and outlines opportunities for future work.

Table 1.1 explains how publications I – IX are mapped to the remaining chapters of this summary.

The chapters sometimes emphasize aspects of the publications that are different from those emphasized in the originals. For example, many of the publications describing client-side assessment include additional contributions which go beyond the scope of this thesis and are not discussed in Chapter 5. I will also provide additional material not present in the original publications but needed to tie my work together.

Table 1.1. To which publications the following chapters (excluding Chapter 7, Conclusions) are mostly based on.

	2 Learning Frameworks	**3 Automated Assessment . . .**	**4 Visualizing . . .**	**5 Assignment Mobility**	**6 Mutation Analysis . . .**
Pub. I: Interaction and Feedback in Automatically Assessed Algorithm Simulation Exercises	x				
Pub. II: Review of recent systems for automatic assessment of programming assignments		x			
Pub. III: Creating and visualizing test data from programming exercises			x		
Pub. IV: Automated Visual Feedback from Programming Assignments			x	x	
Pub. V: Taxonomy of effortless creation of algorithm visualizations				x	
Pub. VI: Adapting moodle to better support CS education				x	
Pub. VII: Two-dimensional Parson's puzzles: the concept, tools and first observations				x	
Pub. VIII: Automatic Assessment of JavaScript Exercises				x	
Pub. IX: Mutation analysis vs. code coverage in automated assessment of students' testing skills					x

2. Learning Frameworks

According to Hill [43], theories describing how people learn are important for two reasons:

1. To give a common framework and a vocabulary to describe learning related observations.
2. To suggest from where to look answers to real learning and teaching related problems.

In addition, Ala-Mutka argues that the use of automated assessment should always be pedagogically justified [3]. Although this thesis is not teacher or learner oriented, it is good for the developers to know in what kind of context their tools are used.

In this section, I will present some learning related theories and point out connections from the theories to teaching programming, assessing programming assignments and providing visual feedback. The rest of this chapter is divided into four sections. Section 2.1 summarizes theories of how learning occurs – is learning something that can be pushed from outside or does it originate from the learners? Section 2.2 uses the Felder-Silverman learning style model to discuss the aspects of automated feedback. Section 2.3 defines the field of program visualization and how learners can interact with visualizations. Finally, Section 2.4 is about measuring the learning.

2.1 Epistemologies

Most of the theories on how learning occurs can be divided between three "Isms" – behaviorism, cognitivism, and constructivism. In the following, I briefly describe these labels with some connection points to programming education.

Behaviorism focuses on observable behaviors. In a way, behaviorism considers educational sciences to be like other natural sciences – response to a stimulus can be predicted based on the previous measurements.

Classical conditioning (e.g. Pavlov's dog experiment) is the extreme of behaviorism. Broadly speaking, any person seeking "hypotheses about psychological events in terms of behavioral criteria" [84] is a behaviorist. This connects behaviorism to many of the evaluation studies used to measure the effectiveness of automatic assessment systems and compare this to a control group not using the tool.

Cognitivism states that learning cannot be forced outside without humans actively participating, thinking, and interacting. Understanding how all this happens is of interest in Cognitivism.

Schemas are a good example of insights cognitivism can bring into CS education. Schemas are abstract plans or other information on how a certain type of a problem can be solved. For example, how to search the "best" element from a list by looping over all values and using one variable to hold the "best" element found so far. An experienced programmer having that schema in his mind can easily spot the schema from a program. He does not need to trace the code line by line to understand what it does. He is also able to write and apply the schema with other schemas, which is essential when writing complex programs. [86, 25]

Caspersen and Bennedsen discuss broadly how to design an introductory programming course based on cognitive load theory, cognitive apprenticeship, and worked examples (a key area of cognitive skill acquisition) [20]. One of their suggestions is to better support learning of schemas. In addition to how to design courses, schemas are related directly to the automatic assessment, for example, through efforts to recognize problem solving strategies automatically from students' programs (e.g. [92]).

Constructivism argues that we all build our understanding of reality by reflecting on our experiences. Students construct mental models to explain their observations instead of storing information poured or provided from outside. Thus, learning is simply adjusting our mental models to the new observations we make. In its extreme, constructivistic epistemology states that a single truth is not a meaningful concept as everyone constructs his own reality.

Today, constructivism is widely accepted and it is widely applied on multiple fields, including teaching natural sciences. However, Ben-Ari, for example, has argued that computer science (CS) is different from the other disciplines. Viable and "correct" mental models of a computer, program execution, etc. do exist and students do not have these models in the beginning. Therefore, although students construct their own knowledge, it is important to explicitly teach these models to ensure students have viable mental models at the end. [10, 9][1]

Active research on students' mental models is inspired by constructivism. For example, Ben-Ari's worry about students not getting the viable models (if not explicitly taught) is experimentally supported by Linxiao et al. [63]. Authors observed that on an introductory Java programming course, two thirds of the students held a viable model of value assignment operation whereas only 17% held a viable model of reference assignment operation. To avoid students having non-viable mental models, the authors propose use of visualizations and creating mental conflicts (i.e. situations that force students to see where their non-viable model fails).

The three "Isms" are labels to many more theories – like cognitivism is the label of cognitive load theory, cognitive apprenticeship, and worked examples. Different learning theories can also support each other and be combined together. For example, the portion of non-viable mental models (constructivism) can be reduced by using visualizations (cognitivism) [63].

To create viable models and good schemas, feedback is essential. When exercises are designed well, automatic assessment can help students to identify non-viable models. After all, exercises and assessment are effective ways to direct the learning process [14, Chapter 9]. However, Greening argues that from the constructivist point of view, automatically assessed programming exercises can be too restricting [37, pp. 53–54]:

> Usually, however, the tasks required of the student are highly structured and meticulously synchronized with lectures, and are of the form that asks

[1]This might be case in some other disciplines as well. For example, models of an atom needs to be taught just like the execution model of a program needs to be taught.

> the student to write a piece of code that satisfies a precise set of specifications created by the instructor. The desired product outcome is typically so trivial and predictable that it makes sense to have students submit their work for automated marking as a matter of convenience. [. . .] Although some practical skills are certainly gained, the exercise is essentially one of reproduction.

Yet, automated assessment of programming exercises is widely used and good experiences have been reported in surveys [3, 19, 28]. Moreover, closed, small assignments are often justified when students start to learn programming. Indeed, automated assessment is often used on beginner and intermediate levels and less later when design aspects of programming, for example, are more important.

2.2 Felder-Silverman Learning Style Model

Learning styles vary. For example, some students prefer facts and hard data before theories. Others prefer visual information, i.e., pictures and animation, before written or spoken information. Some like individual studying and others might prefer interactive learning in groups.

The purpose of learning styles, or learning models as they are sometimes called, is to identify and classify different learning preferences. A learning model can be used for designing a course to meet the needs of different students better.

Learning style models are controversial and the related studies have been criticized by many. Trying to identify learning styles and label students based on that is considered harmful. In addition, the validity of the learning style assumption (e.g. some learners learn better visually) is doubtful and results of various evaluation studies are contradictory. Pashler et. al conclude this by saying [76]:

> The contrast between the enormous popularity of the learning-styles approach within education and the lack of credible evidence for its utility is, in our opinion, striking and disturbing.

Despite the criticism (or as the criticism states), learning styles are widely applied in many fields of education, including computer science education (see e.g. [60, 99]). Although the existence of learning styles as part of personality is questionable, people have opinions and preferences how they would like to learn [76]. Following these preferences can increase motivation. However, my main motivation to introduce learning models here is that they provide a framework for the developers of assessment systems so that they can think how to provide versatile feedback – as discussed in Publication I.

While we were writing Publication I, we were not aware of all the problems of learning style models. For example, we stated that:

> We strongly believe that incorporating analysis of learners' preferences into design of courses, automatic feedback systems, and learning environments leads to better learning.

Based on Pahsler et al.[76], this is likely not true. Thus, I will not use learning models to label students but to discuss different options in automated feedback.

In the following, I describe only the Felder-Silverman learning model as an example. This model was selected because of its popularity in the context of CS education. The variety of possible learning models is huge. Coffield et al. lists 50 different learning models grouped into five categories [22]: stable personality types (e.g. Myers-Briggs [69]), flexible learning preferences (e.g. Kolb [58] and Felder-Silverman [33]), constitutionally based learning styles, styles reflecting deep seated features of cognitive structures, and learning approaches/strategies. Of these, at least Felder-Silverman, Kolb's, and Myers-Briggs's learning models are often applied in CS education.

The Model

The Felder-Silverman model characterizes students' learning styles by using the following dimensions, each of which has two extremes.

Sensory vs. Intuitive – i.e., what type of information does the student preferentially perceive. Sensing learners are practical and oriented toward facts. They typically like straightforward things like working with details or memorizing data. Methodologically they like experimentation and problem solving by standard methods. Intuitive learners like conceptualization. They often prefer theories, principles, innovations, complications and grasping new concepts. All these issues include assignments, analysis and feedback of solutions such that require complicated reasoning.

Visual vs. Verbal – i.e., what sensory information is most effectively perceived. Visual learners prefer visual information like pictures before verbal information. In the other extreme verbal learners like written or spoken language.

Active vs. Reflective – i.e., how does the student prefer to process information. Active learners are group workers who learn by trying things out and prefer continuous interaction. Reflective learners first think things through by themselves, i.e. they do reflective observation. The interaction should drive them to rethink their solution anew promoting their need for theoretical understanding.

Sequential vs. Global – i.e., how does the student progress towards understanding. Sequential learners proceed linearly with small steps whereas global learners learn holistically in large steps. Global learners like to get a holistic view

Programming Assignments

Although Publication I focuses on how to apply learning styles on *algorithm simulation exercises*[2], we also did some observations and recommendations related to programming assignments. These are summarized here.

For my thesis, the most interesting observation of Publication I is that many of the popular assessment systems providing feedback from programming assignments support only textual feedback. Visual feedback is rarely used, and when it is used, differences to textual feedback are small. For example, instead of providing purely verbal feedback, VIOPE [96] highlights parts of the code that failed to pass tests. Web-CAT [30] uses a similar approach to highlight lines of the code that the student has not tested[3].

The other common characteristic is that assessment systems analyze only complete solutions submitted by students. Thus, little aid is given

[2]These are programming related assignment where the answers itself are visual and what PILOT [17] and TRAKLA2 [59], for example, provide.

[3]In Web-CAT, part of the grade comes from how well students test their own programs.

during the process itself. Such mode of working suits intuitive learners who already master, at least partially, the concepts and processes involved. Active learners' preference towards continuous interaction is also not supported if the feedback loop takes too long. Some of the recent tools, however, can address this problem. Marmoset [87], for example, integrates with the development environment and creates an invisible submission whenever the student saves his or her files. Authors have used the logs to study students' behavior but the information could also be used to produce immediate feedback. On some level, smaller assignments may provide a similar effect. This, the size of assignments, is also a good way to support different ends of the sequential vs. global axis because solving large assignments requires a holistic approach. Finally, the amount of formative feedback is often small and interpreting the summative feedback requires reflective processing.

In summary, most assessment systems do not give much support for sensory, visual, and active learners. We must, however, recognize that programming is inherently an activity that requires intuitive, verbal, reflective and global approaches. Writing program code, of course, is an active process but without a strong intuitive understanding of the goals and concepts used and an ability to reflect the results, it is very hard to solve a given programming exercise.

In Publication I we suggest that program visualization tools, as well as visual debuggers should be used more in teaching programming. Such tools can support sensory and visual learners by giving a better insight into what happens while the program is executed. In addition, we should design exercises where active experimentation plays a key role. These could include, for example, studying a working program and preparing data that produce the required output, added with a requirement to reason how the student proceeded to the solution. For sequential learners we could devise a sequence of exercises, where they gradually develop a working program by preparing small incremental changes and additions. Automatic assessment tools can be used to support the process by giving feedback on whether the program is working correctly. Even if such feedback would be merely summative and verbal, the learning process itself can still be designed to support different learning styles.

2.3 Visualizing Software and Interacting with Visualizations

At the end of the previous section we stated that *program visualizations* should be used more with programming assignments. In the next subsection I will define this and some other related terminology. After that, the *engagement taxonomy* [72] describing the ways of how learners can interact with the visualizations will be introduced.

2.3.1 Software Visualization

Price et al. [78] define *software visualization* (SV) as

> the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software.

Authors divide SV further into *program visualization* (PV) and *algorithm visualization* (AV). The distinction is not strict and AV includes parts of PV and vice versa. Often, the difference is understood so that whereas PV can take almost any program and visualize it to enhance understanding, AVs are higher-level representation designed for a specific purpose or even for a specific program, typically related to handling data structures. Moreover, Price et al. separate AV into static visualizations and dynamic *algorithm animations*.

To complement Price's definition, Diehl [26] states that software visualizations can focus on *structure*, *behavior* or *evolution* of software. Structure is about the static properties of a program – control-flow for example. Behavior is about information collected by executing a program with real or abstract input. Visualizing evolution of a program is related to the software development process and how a program changes over time.

2.3.2 Engagement taxonomy

Before going into details of how to implement visualizations, it is good to discuss how students can perceive or interact with the visualizations. For this purpose, I present the engagement taxonomy [72] and the extended version of it [70]. The original taxonomy is based on six levels of interaction: *no viewing*, *viewing* without control of what is viewed, *responding* to questions related to visualization while viewing, *changing* visualization, e.g., by defining the input, and *constructing* and *presenting* visualizations for others. The extended version adds new levels between viewing and responding and between changing and constructing. Finally, it also adds a new level above all the previous. Definitions of the levels of the extended taxonomy, as presented by Myller et al., are provided in Table 2.1. Levels of both versions of the engagement taxonomy are not hierarchical and a single system can support a combination of them.

A meta study of algorithm animation effectiveness, conducted by Hundhausen et al. [45], states that in general, higher engagement implies better learning. The same holds also for the extended taxonomy [70].

The visual feedback proposed in Chapter 1 and discussed more in Chapter 4 provides relatively high engagement. Students *construct* the visualizations though writing code. Although students cannot do much with the image itself, they need to modify their programs to get more feedback (i.e. Construction in the extended taxonomy). However, the downside of visual feedback pointing out problems from students' programs is that only the students who need to submit again (i.e. who did not get full points or enough points in the beginning) will get engaged.

2.4 Levels of Cognition

Teaching goals and levels of learning vary and various taxonomies have been developed to measure this. Such taxonomies can be valuable when designing assignments and also when comparing/grading answers to a specific assignment. Bloom's taxonomy [15] (the revised version of it [8]) and the structure of observed learning outcomes (SOLO) taxonomy [13] are well know examples of taxonomies addressing the level of learning, i.e. cognition. Both taxonomies are also often applied in CS education. Other similar taxonomies are the CS specific learning taxonomy developed by the ITiCSE'07 working group [36] and the recently proposed combination of Bloom's taxonomy and SOLO [65].

Bloom's taxonomy is based on six hierarchical competence levels: knowledge, comprehension, application, analysis, synthesis, and evaluation. Mastery on a certain level requires mastering all the previous levels. More detailed definitions of the cognitive levels and examples of questions being mapped to the Bloom's levels are provided in Table 2.2. The revised version of Bloom's taxonomy replaces nouns with verbs. The new levels, starting from the lowest cognitive domain, are: remembering, understanding, applying, analyzing, evaluating, and creating. Two highest

Table 2.1. Definitions of the levels of the engagement taxonomy quoted from the article where the extended taxonomy was introduced [70]. Levels marked with an asterisk are present also in the original engagement taxonomy.

Level	Definition
No viewing (*)	There is no visualization to be viewed but only material in textual format. For example, the students are reviewing the source code without modifying it or they are looking at the learning materials
Viewing (*)	The visualization is viewed with no interaction. For example, the students are looking at the visualization or the program output.
Controlled viewing	The visualization is viewed and the students control the visualization, for example by selecting objects to inspect or by changing the speed of the animation. [...]
Entering input	The student enters input to a program or parameters to a method before or during their execution.
Responding (*)	The visualization is accompanied by questions which are related to its content.
Changing (*)	Changing of the visualization is allowed during the visualization, for instance, by direct manipulation.
Modifying	Modification of the visualization is carried out before it is viewed, for example, by changing source code or an input set.
Constructing (*)	The visualization is created interactively by the student by construction from components such as text and geometric shapes.
Presenting (*)	Visualizations are presented and explained to others for feedback and discussion.
Reviewing	Visualizations are viewed for the purpose of providing comments, suggestions and feedback on the visualization itself or on the program or algorithm.

levels of cognition are swapped in the revised taxonomy because creating something new is considered to be more demanding than evaluating something that already exists. The revised version also adds the *knowledge*[4] as a new dimension to the taxonomy. The Levels of the knowledge dimension, starting from the simplest, are factual, conceptual, procedural, and meta-cognitive.

Applying Bloom's taxonomy to classify programming assignments can be difficult. Lister and Leaney argue that based on the size of the code, writing code can belong to various cognitive levels of Bloom's taxonomy [62]. In addition, Johnson and Fuller point out even experts are not able to agree about the interpretations of the levels of Bloom's taxonomy [49].

The SOLO taxonomy is based on five levels describing how students' answers can address to what is being assessed. Levels of the SOLO taxonomy, starting from the simplest, are pre-structural, uni-structural, multi-structural, relational, and extended abstract. A pre-structural answer misses the point of the question or is simply wrong. A uni-structural answer names one item from a list of possible topics to be included in a good answer. A multi-structural answer improves from this by providing a list of (uni-structural) answers but the items are still not connected and values of different items are not discussed. These flaws are not present in a relational answer where each part of a multi-structural answer is connected to the whole. Finally, extended abstract, being the highest level in SOLO taxonomy, generalizes and makes connections outside the scope of the question. Meerbau-Salant et al. have combined both Bloom's taxonomy and SOLO to create a new taxonomy and argue that SOLO's holistic vs. local perspective makes it almost orthogonal with Bloom's taxonomy [65].

Taxonomies measuring the level of learning are valuable mostly for teachers. However, I argue that developers can also benefit from these. For example, although teachers design and select the feedback, developers of assessment systems should provide interesting and justified options from where to choose. In this thesis, for example, I develop visual feedback and feedback on testing skills that teachers can try out and evaluate in their own context.

[4]The *knowledge* here is a not the same as the simplest level competence, also called as knowledge in the original version of Bloom's taxonomy.

Table 2.2. Explanations and examples of Bloom's taxonomy, quoted from [88].

Level	Explanation	Sample Questions
Knowledge [1]	The student is expected to recite memorized information about the concept.	"What is a program?"
Comprehension	The student is expected to explain the concept in his or her own words.	"How is a program similar to a recipe?"
Application	The student is expected to apply the concept to a particular situation.	"What is the output of this program?"
Analysis	The student is expected to separate materials or concepts into component parts so that their organizational structure may be understood.	"Create a topdown design for a program to perform a given task."
Synthesis	The student is expected to put parts together to form a whole, with emphasis on creating a new meaning or structure.	"Write a program to perform a given task."
Evaluation	The student is expected to make judgments about the value of ideas or materials.	"Given two programs that perform the same task, which one is better and why?"

[1] Article from where this table is quoted used *Recall* here. However, the original term from the Bloom's taxonomy is *Knowledge*.

3. Automated Assessment of Programming Assignments

Whereas the previous chapter has the focus on educational aspects, this chapter describes the state of the art in automated assessment from the tools perspective. Section 3.1 provides an historical overview to the tools development and explains which features of programs are typically assessed automatically. The section relies on two surveys both published in 2005. Section 3.2 complements the surveys by summarizing Publication II, which has the focus on tools reported between years 2005 and 2010.

3.1 Meta-Survey

This section is divided into two subsections looking assessment platforms from different perspectives. Section 3.1.1 provides a historical overview. This is based on the review of Douce et al. [28]. Based on a survey by Ala-Mutka [3], Section 3.1.2 lists automatically assessed features of programs (i.e. on which automated feedback can be given).

Although there are not many surveys from the field, the selected surveys, while good, are not the only ones. For example, the ITiCSE 2003 working group led by Carter [19] conducted a survey among CS educators (not only programming) to find out how they use assessment tools and what are their opinions towards the use. One interesting finding of Carter et al. and Publication VI is that the teachers who were not familiar with automated assessment considered its potential more limited than the respondents with experience. Finally, there is also a recent survey about programming assessment tools from 2009 [61]. This, however, provides almost nothing new to the previous work of Ala-Mutka.

The number of educational tools (including assessment tools) reported in the literature is high. David Valentine found out that 18% of the papers published in SIGCSE (one of the leading conferences in computing education research) conference between 1983 and 1993 were tools papers, whereas between 1994 and 2003 the number was 24.6% [95]. In the *Survey of Literature on the Teaching of Introductory Programming* by Pears et al. [77] from 2007, tools were the single largest group among papers classified between tools, curricula, pedagogy, and programming languages. Analysis of papers from the ICER, SIGCSE, ITiCSE, ACE, Koli Calling and NACCQ conferences between 2005 and 2008 by Sheard. et al. [85] also supports the importance of both assessment and tools. Top three themes in their classification of programming education related papers were: ability/aptitude/understanding (40%), teaching/learning/assessment techniques (35%), and teaching/learning/assessment tools (9%). Despite automated assessment of programming being only a small portion of all the tools, the number of assessment platforms targeted at programming assignments is still significant (see Publication II).

3.1.1 History of Automated Assessment

In order to create a historical overview, Douce et al. [28] divide assessment tools into three generations. The following list presents the characteristics and some remarkable assessment platforms of each generation. The tools presented next are examples only and the list is not supposed to be comprehensive.

1st generation – *Early Assessment Systems* had very little to build on. They were big, monolithic systems targeted for teachers only. (1960s and 1970s).

Hollingsworth presented the first assessment platform to grade punched cards already in 1960 [44]. Interestingly, security problems related to executing unknown code as part of automated assessment were identified, although not addressed already in 1969 [41].

2nd generation – *Tool-Oriented Systems* are command line based scripts that benefit from small utilities and other services provided by the operating system. Assessing other features than the functionality, and providing feedback directly to students started to become popular. Security of the assessment platforms was also improved. Assessment was often based on character-by-character comparison of the output and the expected output. (1980s' and 1990s)

Examples of popular 2nd generation tools are TRY (1989) [79], Ceilidh

(1993)[12], ASSYST (1997) [46] and BOSS (1998) [51]. TRY introduced the ideas of limiting submission and providing immediate feedback directly for students.

3rd generation – *Web-Oriented Systems* are, as the name suggests, used online with a web browser. Nearly all tools provide some course or content management features. Some of the command line based tool oriented platforms evolved to web-oriented systems. (late 1990s -)

Examples of well-known 3rd generation tools are Course Marker (successor of Ceilidh) [42], Web-CAT (2002) [30] and BOSS.

In summary, automated assessment of programming assignments has been practiced since programming has been taught. In the beginning, feedback was given only on the functionality by comparing the output against the expected one. Today, feedback is based on many aspects of programs as explained next – in Section 3.1.2.

3.1.2 Automated Assessment of Different Features

Ala-Mutka [3] lists features of programs that have been assessed automatically. She divides features between the ones that need execution of a program (i.e. dynamic analysis) and the ones derived from a program code without executing it (i.e. static analysis). According to Ala-Mutka, functionality, efficiency, and testing skills are typically assessed through dynamic analysis. Static analysis, on the other hand, is used to give feedback on programming errors (e.g. dead code), various software metrics, and design. In addition, both static and dynamic analyses are used to give feedback from various special features (e.g. GUI testing).

Functionality evaluates the correctness of programs' behavior. It is the most common automatically assessed feature of programs and nearly all systems are able to give feedback from it. Later on, in Subsection 3.2, I discuss different approaches to test functionality.

Efficiency of computer programs is typically related to the usage time and space (i.e. memory) but the usage resources like disk space, network, or even power consumption of a portable device, can be relevant. In the original survey, Ala-Mutka focused only on the time efficiency, perhaps because it still seems to be the dominant or only efficiency metric supported by automated assessment platforms. There are many options how to present the feedback in this category. For example, CPU time can be plotted as a function of input size as in the algorithm benchmark extension of OpenCPS [21].[1]

Test adequacy [2] of students own tests has become more interesting – perhaps because of industry saying that new graduates lack essential testing skills [29]. Students write tests to their own or some other programs and feedback is then given based on various test adequacy metrics (e.g. [66, 100]). Automated assessment of testing skills was supported already in 1997 by a tool called Assyst [46]. Today, Web-CAT is a widely used tool designed around the principle that students test their own programs.

Style is perhaps the most obvious feature to be assessed through static analysis. In most programming languages, there are (de-facto) standards defining indentation, variable naming conventions and other typesetting features. Although modern IDEs (e.g. Eclipse[3]) make it easier to write well-formatted code, giving feedback from style is still relevant. Quality of comments and related documentation are also part of programming style.

Programming errors includes use of un-initialized variables, dead code (i.e. code that will newer be executed), and other errors detectable by static analysis. Many automated assessment tools incorporate static analysis tools such as Lint family (e.g. JSLint[4]) and Valgrind[5].

Software metrics are easy to generate but the educational motivation needs to be carefully considered each time. For example, statement count, branch count, cyclomatic complexity, lines of code, lines of comments, percentage of lines containing comments, and code depth are useless for students unless accompanied by a desired value or range for the measure.

Design might not be the most obvious automatically assessed feature. Feedback is typically about the details of the design, not really about the high level design. Examples of recent work that could be used in automated assessment of the higher level design are work by Dong

[1]`http://www.opencps.org/`

[2]Original term used by Ala-Mutka is *testing skills*. However, to be consistent, I wanted to use a term that is a feature of a program, not a feature of a person.

[3]`http://www.eclipse.org/`

[4]`http://www.jslint.com/`

[5]`http://valgrind.org/`

et al. [27] to recognize design patterns from programs, and work by Taherkhani [91] to recognize different sorting algorithms through static analysis. Lower level design issues, that are actually used in the existing automated assessment tools (e.g. [94, 81]) check if the structure of the solution matches the pool of allowed structures (e.g. there is a loop or a recursion present).

Special features are language, assignment, and topic related features that do not fit into any of the previous categories. Examples of this category are GUI testing, disallowing some language constructions (e.g. use of `set!` in Scheme [81]), plagiarism detection (e.g. [1, 35]), etc. Some of the special features are best addressed with static analysis while others require executing the programs under assessment.

On What the Feedback is Typically Given
Douce et al. [28] point out that the functionality has always been the most common feature on which automated assessment is provided. Carter et al. [19] lists functionality, efficiency, and complexity as the most common automatically assessed features. Complexity and *software metrics*, discussed earlier, are closely related features.

3.2 Aspects of Automated Assessment

This section is based on Publication II where a *systematic literature review* [16] was carried out to find out the features of automatic assessment systems reported recently in the literature.

In Publication II we applied an iterative process to find a consensus about how to group features of the systems. We read a set of papers, made the first version of categories, read more papers, revised the categories, and went through the previous papers to find if something related to the new categories were expressed there as well. This was repeated until no new categories emerged. Our background in automated assessment explains some of the results. We (i.e. author of Publication II) have all used automated assessment several years, and we have also been developing assessment platforms and other educational software to support CS education.

The next subsection will present the categories we found in our survey and Section 3.2.2 will conclude this.

3.2.1 Features of Automated Assessment Platforms

Programming Languages

A majority of the systems identified in Publication II are either targeted primary for Java or at least support Java. This fits well with the trend of Java being one of the most used introductory programming languages. Other popular languages supported by the systems are C/C++, Python, and Pascal. Pascal is no longer a common teaching language and we were surprised that some assessment platforms, originating mostly from the Eastern Europe, were targeted for Pascal. Examples of other supported languages are Assembly and shell scripts.

It should be noted that some of the systems are language independent. Especially, if the assessment is based on output comparison, any programming language that can be executed on the assessment platform can also be assessed by the platform.

Learning Management Systems

Integrating CS specific features such as the automated assessment of programming assignments into learning management systems (LMS) has been a popular topic for a long time. Publication II identifies several fully automatic assessment tools supporting programming assignments built on Moodle[6] (see [5, 38, 48]), Sakai[7] (see [89]), Cascade LMS[8](see [40, 74]), and Plone[9] (see [6, 7]).

One argument of integrating assessment tools and LMSs is to avoid reimplementing course management features in both systems. However, it is not clear if separate LMSs are truly needed because some of the assessment systems are capable to grow into the role of a LMS. For example, Web-CAT, with the various assessment modules already implemented into it, is a good candidate to become a CS specific LMS.

Executing unknown programs is always a security threat. A typical LMS installation hosts several (not only programming) courses. Malicious code executed in such an environment can harm all of the courses hosted on the system. Therefore, securing assessment systems integrated into LMSs is extremely important.

[6] http://moodle.org/
[7] http://sakaiproject.org/
[8] http://www.cascadelms.org/
[9] http://plone.org/

How Tests are Defined

Testing the functionality is the most common approach to grade programs, and output comparison is a common approach to do that. Survey of Ala-Mutka [3] already reports several variations of output comparison based grading, including running the model solution and student's code side by side, and use of regular expressions. In addition, many tools have been borrowed from the software testing industry. For example, XUnit family and web testing frameworks such as Watir[10] and Selenium[11](e.g. [23, 90]) are often applied to grade students' programs.

Experimental approaches, like comparing program graphs to a pool of correct programs, have also been suggested [73, 98]. However, these have not gained wide popularity.

Resubmissions

Exercises should have room for mistakes and learning from them. However, to prevent mindless trial-and-error problem solving, the number of resubmissions should be somehow controlled [64].

Different approaches to tackle the problem of resubmissions were identified in Publication II. In summary, these can be divided among limiting the number of submissions, forcing a time penalty after each submission, making each trial slightly different, having only a very limited time window when exercises are open (i.e. contest style), and various combinations of these.

Possibility for Manual Assessment

It is often a good idea to combine manual and automated assessment. For example, teaching assistants (TAs) can provide extra feedback to a manually assessed submission. To enable TAs and teachers to view students' submissions is the lightest way to support for manual intervention. Supporting this through the automated assessment systems makes it possible to separate the roles of TAs and the roles of administrators. Some systems (e.g. Web-Cat [30]) allow combining manual and automatic feedback. This means that TAs' feedback and automated assessment can both exist at the same time and support each other.

[10]http://watir.com/
[11]http://seleniumhq.org/

Sandboxing

Since the programming assignments are typically graded by running the students' solutions on the server side, securing the server against possibly malicious or just incorrect code is important. A good discussion on the possible attacks against a grading server can be found in [34]. However, as important as this topic is, very little attention is paid to it. Too often security is not considered or various ad hoc solutions like using regular expressions to filter out malicious code from C programs are proposed (e.g. [6]).

Many of the tools that pay attention to security are built on top of existing security solutions such as `systrace`, Linux security module, Java security policies, and `chroot`. However, this is often not enough. Security should be enabled by default and the related documentation should be good enough to make sure that teachers will not make shortcuts when installing assessment systems.

An interesting, and perhaps emerging, approach is to do the assessment on the client side. Security in the course management side is guaranteed by pushing the assessment away from the server. However, the new challenge is how can we trust that the assessment on students' machines really does what we expect it to do.

Distribution and Availability

It is surprising, and quite disappointing, to see how few systems are open-source, or even otherwise (freely) available. This might be one of the reasons for the constant development of new tools – that are also likely to remain in-house. In many papers, it is stated that a prototype was developed, but we were not able to find the tool. In some cases, system might be mentioned to be open source but you need to contact the authors to get it. Publication II argues that by open-sourcing the existing tools to some popular online version control repository like GitHub[12] or Google Code[13], the tools would be much more widespread and more willingly adopted by others.

Exercise Topics

Assignments can set up special requirements for automated assessment. Correspondingly, some assessment systems specialize in very specific types of assignments. Fields of specialization, mentioned in Publication II, are

[12]`http://github.com/`

[13]`http://code.google.com/`

graphical user interfaces, databases/SQL, concurrent programming, and web programming. Modular assessment systems are preferable so that a specialization can be implemented as a plugin to an existing system, rather than implementing yet another assessment system.

3.2.2 Conclusions

The number of different assessment platforms reported in the literature is large. Researchers and lecturers of the universities where the tools are used develop many of these tools. Instead of constantly producing new assessment platforms, institutions should collaborate more and increase the adoption of the existing platforms. I argue that this kind of collaboration would be easiest to establish if the platform itself is open source.

In Publication II we predicted that the importance of integrating automated assessment into learning management systems (LMS) (discussed more in Section 5.2), usability, single-sign-on, and assessment of web programming becomes higher. Interestingly, LMS integration and interoperability (i.e. single sign on in our survey) were pointed out as future directions also in the earlier survey by Douce et al. [28]. What we did not explicitly report in Publication II, but what clearly exists, is the lack of visual feedback – especially visual feedback related to functional behavior. This will be the theme of the next chapter.

4. Visualizing Data Structures for Feedback

The question of *how to provide visual feedback from automatically assessed programming assignments*, was raised in Chapter 1 and will get operationalized in this. The rest of the chapter is divided into five sections. Section 4.1 will refine the scope where the results of this chapter are valid. Section 4.2 will present the main idea of visual feedback. Section 4.3 will present two research problems related to decreasing the level of details in visual feedback and evaluating the usefulness of the feedback. These problems will get answered in the last two sections of this chapter.

4.1 The Refined Scope

As explained in Section 2.3.1, software visualizations can focus on *structure*, *behavior* or *evolution* of software. My focus in this chapter is to provide visual feedback on the *behavior*. This means automated assessment of *functionality* (see Section 3.1.2 for other possibilities).

Moreover, I limit myself to test-based assessment, which means that a test itself is a program that executes and examines students' programs. A test program is often built using a testing framework such as JUnit. The use of such frameworks, however, is not necessary. Although some of the ideas presented later can be extended to procedural languages with memory pointers in general, I focus on testing object-oriented programs.

4.1.1 How Test-Based Assessment Works

The main idea of software testing is simple: the program under test is executed with some input (i.e. test data) and the consequences of executing the program are then compared to the specification. Confidence that the program is correct is gained if observations do not differ from the specification.

```
 1 public class BstTests {
 2    public void testDelete() throws Exception {
 3       BinarySearchTree bst = BinarySearchTree();
 4       bst.insert(1);
 5       bst.insert(3);
 6       bst.insert(2);
 7       bst.insert(4);
 8       bst.delete(3);
 9       assertEquals(2, bst.getRight().getData());
10       assertEquals(4, bst.getRight().getRight().getData());
11    }
12 }
```

Program 4.1. An example how to test BinarySearch tree where students implemented a delete operation.

The program under test is typically either a single method or a set of methods. Arguments passed to the methods under test are the test data. Program 4.1 demonstrates what a test looks like. Depending on what methods are provided and what methods students are asked to implement, the example tests either one method only or a sequence of methods (i.e. inserts and delete). In the latter case, the test data consists of the empty binary search tree in the beginning accompanied with the values inserted and deleted. However, if we can assume that the insert-method has no defects, then the lines from 3 to 7 are test data construction. In this case, the program under test is the delete-method.

A common way to write tests is to call a method and assert something after that, e.g. examine how the method under test changes the object or object hierarchy. Internal behavior of a method can also be tested. To do this, test data passed to a method can contain mock objects that check that they are handled as expected. For example, mock objects can check that some pre-defined methods of them are invoked or unexpected parts of the test data are not accessed. Libraries such as JMock[1] can be used to create mocks with such assumptions. However, based on my own experiences, teachers often write the mocks by themselves to ensure that the error messages and therefore the feedback better fit the needs of novices.

[1]http://www.jmock.org/

4.1.2 Connections to Model Based Testing

A significant difference between testing students' programs and industrial software testing in general is that with students' programs there is typically a fully functional model solution that can be compared against students' programs.

In a way, testing students' programs is very similar to *model based testing* where a simplified model with the desired functionality is first implemented. Tests, including inputs and assertions, are then derived from the model. In model based testing, assertions are based on observing the behavior of the model that is often on a different abstraction level than the actual program. For example, a model of a multiplier can say that the output of multiplying any two numbers with the same sign is positive. If the signs differ the output is negative and if any of the multipliers is zero, so is the output. It is now possible to give data (e.g. random data) to the model and use the output to create the assertions of what should happen when the same data is passed to a real multiplier. The previous example is deliberately simple to demonstrate how the abstraction level of the model and actual solution can differ.

It can be argued that in automated assessment the abstraction level of the *model* (i.e. written by a teacher) and *programs to be tested* (i.e. written by students) are the same. However, the level of abstraction in the model solution that can be applied in testing is not finer than in the assignment description. Internals of the model solution can, and most likely will, be different from students' programs.

Program 4.2 demonstrates how a test in an educational setup can be build on top of a reference solution. The test first constructs two identical binary search trees – assuming that the insert method of both trees is either provided by the teacher or tested beforehand. After initializing the input, the test tries to delete the same key from both trees and asserts that the trees – model solution and student's solution – are equal.

4.2 Visual Feedback

Static data structures (e.g. test data and expected output) can be represented with manually constructed images – similar to Figure 4.1. The example figure illustrates the test data of Programs 4.1 and 4.2. From a technical perspective, visualizations of dynamic *input* and the differences

```
public class BstTests {
    public void testDelete() throws Exception {
        BinarySearchTree bst =
            new StudentBinarySearchTree();
        BinarySearchTree model = new ModelAnswer();
        bst.insert(1); model.insert(1);
        bst.insert(3); model.insert(3);
        bst.insert(2); model.insert(2);
        bst.insert(4); model.insert(4);
        bst.delete(3); model.delete(3);
        assertTreesEqual(model, bst);
    }
}
```

Program 4.2. An example how to test BinarySearch tree where students implemented a delete operation.

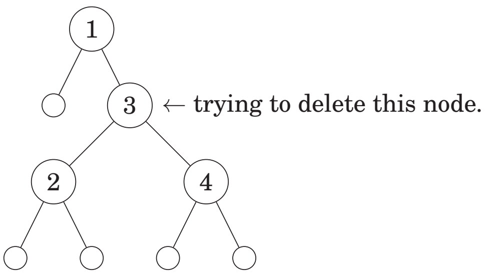


Figure 4.1. Test data visualization related to Program 4.2

between the expected and actual output are, however, more interesting. Those cannot be constructed before tests are executed. Figure 4.2 provides a sketch of what this kind of feedback could look like in our previous binary search tree example. One diagram is needed to visualize the value of each variable of an object type – including the state of the current object. This implies that visualizations of input, expected output, and actual output can each consist of multiple object diagrams as in Figure 4.2.

4.3 The Problems

Now that we have sketched what we would like to achieve, it is possible to ask how to implement this. Visualizations presented previously in this chapter are essentially object diagrams[2] – although with a "pretty" layout.

[2]Object diagram is the visualization of an object graph.

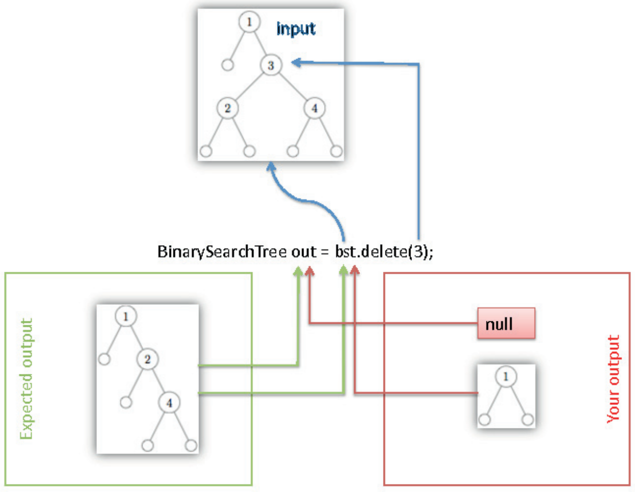


Figure 4.2. Visual Feedback of input, expected output and actual output. Test tries to delete ’3’ from the tree. The correct data-structure after the operation is in the lower left corner. On the right is the output of the student’s program. It deleted the whole subtree starting from the node where ’3’ is. The program also returned *null*-reference instead of the modified data structure.

The state of an object (i.e. object graph) is easy to capture. To get good visualizations and meaningful results from the comparison of two object graphs (i.e. assessment based on the comparison against the model solution), the granularity and abstraction level of the object graphs should be controlled. This is because, not all data are useful when graphs are compared or visualized. In addition, some educational results indicate that abstract feedback forcing students to think is preferable to too detailed feedback, which may make students passive [67]. Thus, the research question we will look next is:

> How to raise the abstraction level of object diagrams that are used to provide visual feedback?

Methods to address this problem are presented in Section 4.4. Another problem related to the usefulness of visual feedback, discussed in Section 4.5 is:

> What is the value of visual feedback for learners and for the teachers?

To study this, a small control group experiment comparing the performances of groups getting different feedback was carried out and described originally in Publication IV.

4.4 Controlling the Level of Detail in Object Graphs

Typically, not all parts of an object graph are interesting. For example, in our previously presented binary search tree example, quite likely only the references pointing to the child nodes and data are relevant. If a student stores some extra information in a node, this information is not needed in the assessment and therefore needs not to be extracted. In addition, sometimes limiting the depth of an object graph is important to ensure that the amount of information presented is manageable. For example, internals of a StringBuffer object, provided by the standard Java library, are rarely interesting in the feedback.

A Lightweight Java Visualizer (LJV), introduced by Hamer in 2004 [39], allows users to specify which of the outgoing references for each object type are followed when constructing an object graph. It is also possible to control, if some object combinations should be folded together, instead of presenting each object belonging to the specified combination as a separate node in the object graph. After extraction, LJV can visualize the graph with Graphviz[3].

In Publication IV, we implemented a visualization system similar to LJV. In addition to presenting two object diagrams (i.e. model and measured outputs), our tool also highlights where the two graphs differ. An example of this is provided in Figure 4.3.

A different approach to raise the abstraction level was selected in Publication III, which is summarized next.

4.4.1 Generalized Symbolic Execution With Lazy Initialization

One of the key ideas in model based testing is that tests are derived from the model. In our previously presented model based automated assessment example (i.e Program 4.2), only the assertions were derived from the model. The test data were constructed manually.

Symbolic execution [57] (and an extension of it called generalized symbolic execution with lazy initialization [56]) can automate test data gen-

[3] `http://www.graphviz.org/`

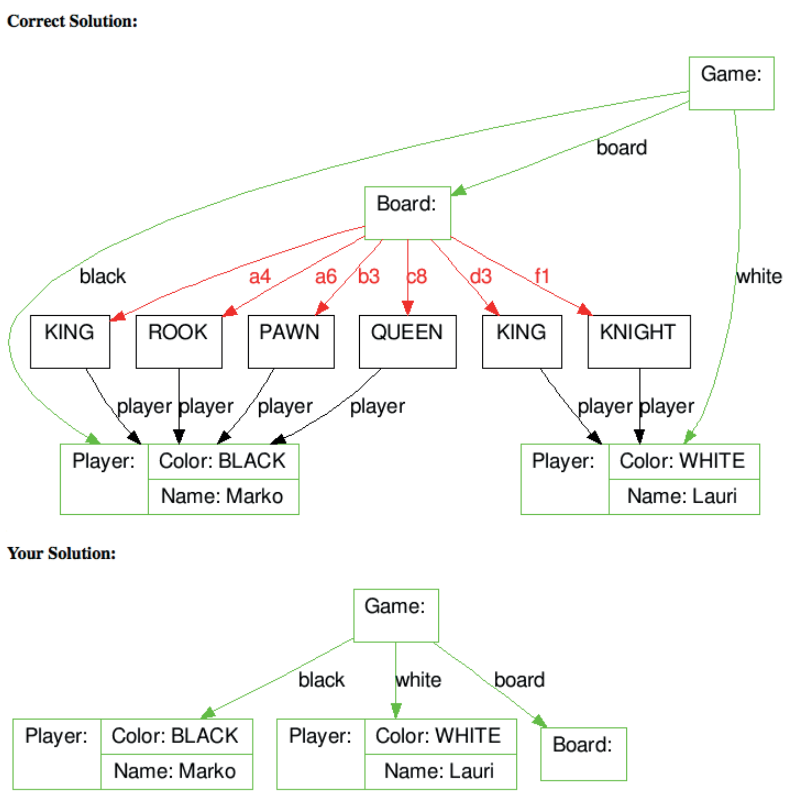


Figure 4.3. Visual feedback of output data structures based on automatically extracted object graphs. Graphs (model solution and student's solution) are compared and the parts that are found matching are colored green. Differences are colored red. If some parts are not compared, because of earlier differences, these are black. (figure originally published in Publication IV)

eration. An interesting side effect, discussed in Publication III, is that the abstract state from where each input is derived, can also be used to visualize and highlight the relevant parts from the test data.

A program state in symbolic execution consists of (symbolic) values of the program's numeric variables, a path condition, and the program counter (i.e. information where the execution is in the program). Path condition is a boolean formula over input variables and describes which conditions must be true in the state. This can be explained best with the following example where the symbolic execution of Program 4.3 is explained.

The hierarchy of all the symbolic states (i.e. symbolic execution tree) of Program 4.3 is presented in Figure 4.4. The data inside each node are the values of variables and the path condition (PC). A number above each node denotes the line number that is going to be executed next. A state underneath a line number, however, is the state after the line is (symbol-

```
1 public static int min(int a, int b) {
2   int min = a;
3   if ( b < min )
4     min = b;
5   if ( a < min )
6     min = a;
7   return min;
8 }
```

Program 4.3. A program calculating the minimum of two arguments. Line 6 is dead code (i.e. never executed) as one can see from Figure 4.4.

ically) executed. It should be noted that each reachable conditional in the program leads to branching in the symbolic execution tree.

The middle node on the fourth level (first level being the topmost) of the tree is interesting. The path condition at the end of the line is infeasible (X<X), which means that no real execution can reach this state. Such states, where no real execution can go, are marked with "backtrack" in the figure. Leaf nodes with feasible path conditions provide all the possible execution paths of the program. Thus, we can immediately see that the program has only two possible execution paths. Input where a>b (i.e. a:X, b:X, Y>X) results in execution of lines 1, 2, 3, 4, 5, and 7. The node in the lower right corner is the other possible symbolic end state of the program. It is possible to follow the nodes leading to the leaf node to find out what the execution path is. That is, with a $\leq$ b, lines 1, 2, 3, 5, and 7 are executed – in this order.

Generalized symbolic execution with lazy initialization defines that when an unused (no previous reads or writes) field of a primitive type is accessed, it is initialized to a new symbolic variable. Whenever an uninitialized field of a reference type (i.e. object type) is read, it is nondeterministically initialized to any of the following:

- `null`
- a new object with uninitialized fields
- a reference pointing to any of the previously created objects of the same type (or subtype)

This, however, can lead into illegal states – for example by creating a cycle to a data structure where loops are not allowed. To prevent this, a method to identify states that cannot be completed to legal structures is

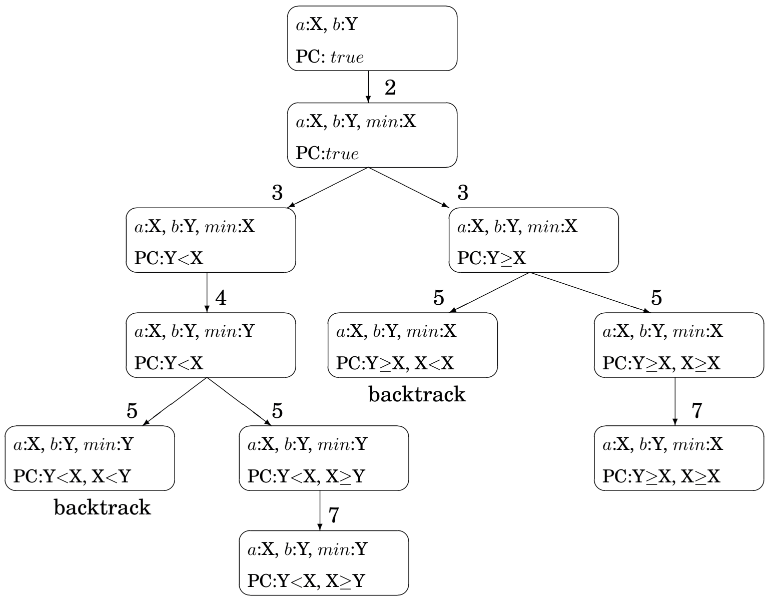


Figure 4.4. Symbolic execution tree of the Program 4.3. Numbers in the figure are line numbers. (figure originally published in Publication III)

needed and called after each lazy initialization. This method is specific to the topic at hand and in some cases difficult to implement well.

Often test data are not only a list of primitive values but consist of an object hierarchy. In both cases, it is possible that only some parts of the data (e.g. object graph) are relevant to the execution. This means that when the program is executed, some parts of the data are not read and therefore the values do not affect the execution.

When generalized symbolic execution with lazy initialization is used to produce the test data, some of the references can be found un-initialized in the beginning. This means that the corresponding values can be anything structurally legal. When visualizing test data, this can be abstracted away or a visual hint advising that those parts are not relevant can be provided. An example of this is provided in Figure 4.5 where subtrees drawn as triangles denote parts that are not accessed by the program. Circular nodes with a question mark inside are accessed but the data stored in the node are not. Symbolic constraints related to the input are presented underneath the graph.

If the program under test lacks functionality and therefore is structurally simpler than the model program, input generated from the incomplete implementation will likely not test the missing functionality. To fix this, manually created sanity tests, random input or input generated from the model solution should be used. However, abstract and only partially initialized input structures generated from the model solution should be completed (i.e. uninitialized references should be opened up) before they can be used. This is because the program under test can access the objects differently when compared to the model solution.

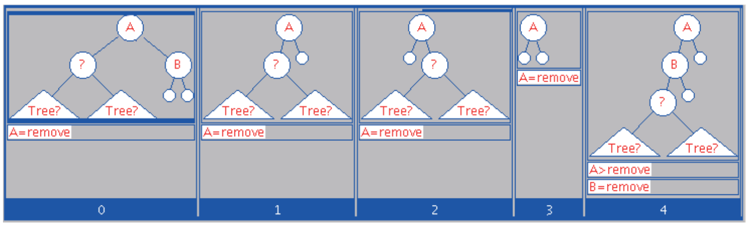


Figure 4.5. Excerpts of partially initialized input structures for the delete method of binary search trees. (figure originally published in Publication III)

A prototype to demonstrate symbolic visualizations was implemented in Publication III, but we did not integrate it into any automated assessment platform. The prototype was built on Java PathFinder (JPF) [97] and the symbolic execution framework provided with JPF at that time.

4.5 Evaluation Study

In Publication IV, we studied the effect of visual feedback on students' performance. We divided students into three groups and gave different feedback from different exercises for each group. Feedback was given based on:

- automatically extracted object graphs where the differences between the expected and the correct outputs were highlighted (see Figure 4.3 on page 37).
- exercise specific visualizations where the semantics of the data was taken into account when designing the presentation (see Figure 1.2 on page 4).
- specific textual feedback. Because the same assignments were almost unchanged from the previous years, this feedback had matured and was of good quality.

We investigated properties like the number of submissions, time needed to do a resubmission, and correctness of best submissions. From any of these properties, we were not able to find statistically significant differences between the groups.

Based on our experiences, writing high quality verbal feedback is not easy and requires substantial amount of work. Accordingly, designing good custom visualization can also take time. Object graph based visual feedback, however, requires less effort from the teacher to prepare. Thus, as argued in Publication IV, visual feedback not performing worse than good textual feedback is a promising result. Teachers may end up spending less time to produce feedback that seems to be equally valuable for the students.

Ben-Bassat et al. [11] have pointed out that visualizations can help by providing a common vocabulary and model to better understand program execution. The authors also point out that *“the interpretation of the animation itself is non-trivial and must be explicitly taught.”* Feedback from our students supports both observations. Although some students found visualizations helpful, some complained that visualizations were hard to understand. It would be interesting to see if teaching students to read the feedback would lead to even better results.

5. Assignment Mobility

Many CS educators are willing and share programming assignments but there are (technical) obstacles to prevent this. Sharing is especially problematic if assignments are assessed automatically [31]:

> Most existing systems present a kind of “black hole” effect – once you configure and set up your assignment, there is no effective way to export or share your work with others or perform incremental updates easily.

Edwards et al. have proposed an open format shared by the assessment platforms to ease the problem [31]. Unfortunately, most assessment platforms do not use any such format.

The approach I selected is to seek answers to the following, perhaps easier, problem of *how to allow assignment implementations and visual feedback to be more independent from assessment platforms*. The rationale behind the question is to make assignment implementations easier to port from one assessment platform to another or even let assignments to work standalone.

The rest of this chapter is divided into four sections. The first three each provide a slightly different perspective to how to support assignment mobility. The goal of Section 5.1 is to promote understanding of what features of visualization tools affect the effortless use of them. Understanding these requirements can help when selecting visualization tools to be integrated into assessment platforms. Section 5.2 explains how Moodle, a general purpose learning management system, could help sharing programming assignments. Section 5.3 presents three case studies of doing assessment (or visualizations) on students' browser instead of on the assessment server. Finally, Section 5.4 concludes this chapter.

5.1 Effortless Creation of Algorithm Visualizations

Lack of effective development tools is one of the main reasons for teachers not to adopt algorithm visualization (AV) tools [72]. Thus, our first goal was to understand what features affect to the effortless use of the tools and therefore what features of tools should be considered if visualization tools are integrated into automated assessment.

We started our research from the simple observation that tools designed to a very specific need are often more effortless when compared to more generic AV systems [54]. Based on this, we conducted a questionnaire for CS educators to identify the typical use cases for AV systems [55]. Finally, in Publication V, we combined the results of our survey with the related research done by others and proposed a Taxonomy of Effortless Creation of AV. The taxonomy is based on the three categories presented next.

Scope

Category Scope is basically defined in how wide a context one can apply the system. The classification is based on four levels: *lesson-specific*, *course-specific*, *domain-specific*, and *non-specific*. For example, when a particular AV system is applied in education, it can be used, for example, during a single lecture, throughout the course or in multiple courses.

Integrability

Category Integrability lists the features that support the system to be easily integrated into an educational setup. In the article, we presented the following requirements related to the integrability: *easy installation*, *customization*, *platform independence*, *internationalization*, *documentation*, *interactive prediction support* (e.g. stop-and-think questions in JHAVE [71]) *course management support* and *integration of hypertext*. However, we also recognized that the list changes rapidly and the meaning of many features changes when the time goes on. For example, easy installation today is not the same what it meant five years ago.

Interaction

Interaction category is divided between *producer-system* (PS) interaction and *visualization-consumer* (VC) interaction as illustrated in Figure 5.1. We applied a two-dimensional classification to measure the producer-system interaction – the first dimension indicating the task (e.g. preparing lecture slides), and the second measuring the time related to the task. To characterize the visualization-consumer interaction, we applied the engagement taxonomy (see Section 2.3.2).

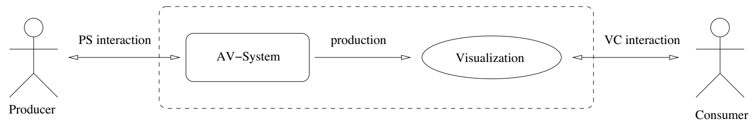


Figure 5.1. Two types of interaction: Producer vs. AV system and Visualization vs. Consumer (figure originally published in Publication V)

5.1.1 Conclusions

Understanding the requirements of effortless use of algorithm visualizations can help when selecting visualization tools to be integrated into assessment platforms. The goal of the taxonomy presented in this section is to promote understanding of this.

In Publication V, we applied the proposed taxonomy to four different, actively developed open source AV systems: Animal [80], JAWAA 2 [2], Jeliot 3 [68] and MatrixPro [53] and concluded that there is need for systems with broader scope (i.e. more generic) that are also effortless to use. We also concluded, that the interaction techniques seem to play the key role in the overall effortlessness (or lack of it). The trend is towards more versatile and interactive AV systems that support several interaction techniques (e.g. programming and direct manipulation) and pedagogical contexts (e.g. exercises).

Since Publication V was written, many new visualization tools have been developed. What is, however, more important is that the systematic collection of visualizations and visualization tools has also been started. The portal for this effort is called Algoviz[1]. Thus, to select a visualization tool, I suggest searching from Algoviz and applying our taxonomy to think what features are important for your specific needs.

5.2 Moodle and Programming Assignments

Although many universities already use learning management systems (LMSs), for programming assignments they still often set up a separate platform. However, if an automated assessment platform would be a plu-

[1] `http://algoviz.org/`

gin of an LMS, then the users of the same LMS could share assignments simply by installing the plugin. This would support assignment mobility because popular LMSs are already more popular than assessment platforms used to give feedback from programming assignments. In addition, having one backend for different courses (not only CS or programming) reduces the need to duplicate LMS functionalities in assessment tools.

Moodle is one of the most popular open source general purpose learning management system. One of the ITiCSE 2010 working groups, which I also participated, investigated how Moodle supports teaching CS (Publication VI). As part of this study, we also wanted to find out how automated and semi automatic assessment of programming assignments are supported and how teachers would like those to be supported. The study was based on questionnaires sent to SIGCSE mailing list and surveying potential Moodle modules we were able to find out.

We concluded that mainstream LMSs in general, and Moodle in particular, do not yet provide adequate support for automatic code management, assessment, and visualizations. Although the level of support in these areas is not satisfactory, there are some interesting tools with good potential – some of them being listed in Publication II. In addition, as already mentioned in Section 3.2.2, it is quite likely that in the future we can see more assessment platforms being integrated to Moodle as there are certainly many teachers interested in trying such tools.

5.3 Browsers' Responsibilities – Case Examples

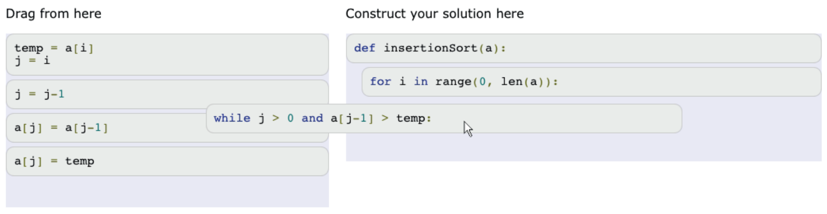
Most assessment platforms do not take the full advantage of the browser environment yet. Today, web browsers are powerful environments with good programming support and *fat clients*, where functionality is shifted from servers to clients, has been a strong trend in web application development [47, 93]. Unfortunately development of learning tools is slightly lacking behind this. Interestingly many of the new visualizations and visualization tools reported in Algoviz are written in HTML/JavaScript and thus the visualization field is catching up.

The reason why fat-client approach is important for assignment mobility is that if automatically assessed assignments are implemented so that a browser can do nearly everything, this implies that there is very little assessment platform specific code to be ported when assignments are moved to a new server environment. In the following subsections, I describe three examples of doing assessment at least partially on client side and explain how this promotes assignment mobility.

5.3.1 Two-dimensional Parson's Puzzles

Parson's programming puzzles are simple assignments where the lines of a program are given in wrong order and the task is to construct the program by sorting the lines [75].

In Publication VII, we described a new family of Parson's puzzles where lines of a Python program need to be placed to a 2-dimensional grid. Puzzles, just like Python programs, use horizontal positioning (i.e. indentation) to define the block structure of a program. This allows creating short but still challenging puzzles. Figure 5.3 illustrates what our puzzles look like.



```
temp = a[i]
j = i
```

```
j = j-1
```

```
a[j] = a[j-1]
```

```
a[j] = temp
```

```
def insertionSort(a):
  for i in range(0, len(a)):
```

```
while j > 0 and a[j-1] > temp:
```

Figure 5.2. User solving two-dimensional Parson's puzzle (figure originally published in Publication VII).

We have also implemented online tools that allow puzzles to be created, shared and embedded into web sites. Puzzles are standalone JavaScript widgets that can do all the assessment inside a browser. Thus, if puzzles are used for self-study purposes, there is no need for a separate assessment server. Puzzles can also log the users' actions (intermediate steps of how puzzles are solved) and send the logs together with the outcome of the assessment and final solution to the server. This helps teachers to understand what their students are doing and where they might have problems. Puzzles are open source and new assignments are easy to define. Tools developed in this project include a collaborative environment[2] where teachers can create, share, combine, and deliver puzzles to their students. The objective of this site is the same as in Greenroom [18] – to support teachers in sharing the study material.

Each assessment platform has a pipeline to handle submissions. This may include unpacking a zip file to a sandbox environment, running some

[2]http://www.parsonspuzzles.com/

scripts to produce the feedback, copy the results to the database and inform student's client that the results are ready. Strictly defined pipelines of how to do assessment can raise problems when integrating assignments with client-side assessment to assessment platforms. Thus, the server-side communication mechanism we implemented for our puzzles is slightly different. Because the feedback is already produced, the server should only store the results. In many cases, especially if the assessment server relies on modern web development frameworks (e.g. Ruby on Rails or Django), defining a new pipeline is easier than adopting to the existing pipeline. Therefore, each submission from our Parson's widget is JSON formatted data (i.e. feedback and logs) submitted to the URL defined in the assignment. To integrate puzzles into a new server environment, programmer needs to understand where the results should be finally stored and implement a separate handler for submissions of self-standing assignments. Implementing this is especially simple if the web framework provides user management.

The limited nature of Parson's puzzles makes them clearly a special case. Assessment is straightforward because students' programs are not actually executed on client side. However, client-side assessment of real programming assignments will be discussed in the next subsection.

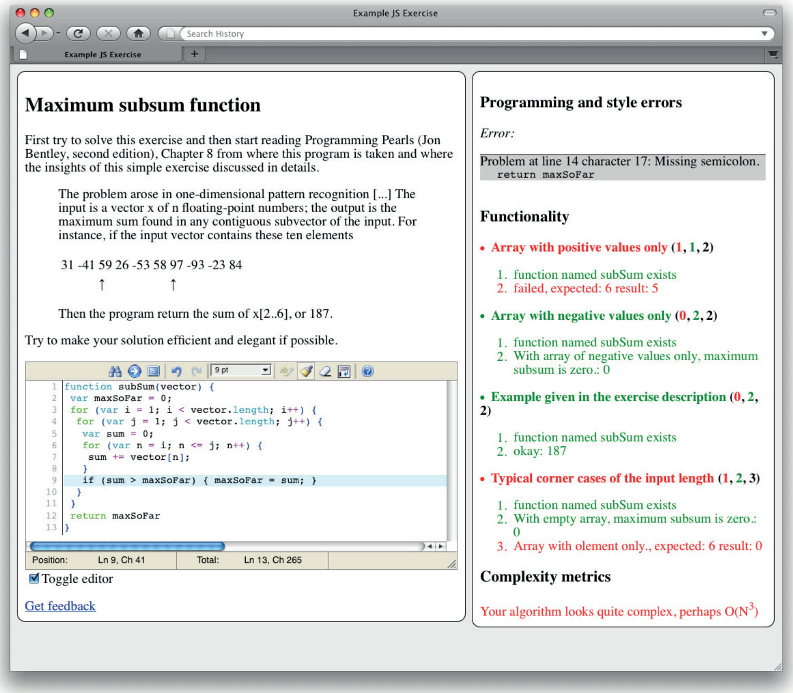
5.3.2 Testing Programs in Browsers

In Publication VIII we implemented assessment of JavaScript programming assignments on client side. Just like the previously introduced set of tools, this work is also open source.

First, we surveyed several client-side JavaScript development tools to find out how those can be adopted to provide feedback from programming assignments. Based on the survey, we implemented js-assess[3]. This tool demonstrates how industrial tools can be integrated to give feedback from functionality, style, programming errors catchable by static analysis, and software metrics. These assignments, just like the Parson's puzzles, can be embedded in any webpage with little effort. Figure 5.3 illustrates what the feedback from our tool looks like.

Although our point of view in Publication VIII is in teaching Web programming in general and JavaScript in special, the idea of implementing in-browser-assessment by decomposing other testing tools is at least par-

[3]`https://github.com/vkaravir/jsassess-demo/`



Maximum subsum function

First try to solve this exercise and then start reading Programming Pearls (Jon Bentley, second edition), Chapter 8 from where this program is taken and where the insights of this simple exercise discussed in details.

> The problem arose in one-dimensional pattern recognition [...] The input is a vector x of n floating-point numbers; the output is the maximum sum found in any contiguous subvector of the input. For instance, if the input vector contains these ten elements
>
> 31 -41 59 26 -53 58 97 -93 -23 84
>
> ↑ ↑
>
> Then the program return the sum of x[2..6], or 187.

Try to make your solution efficient and elegant if possible.

```
function subSum(vector) {
 var maxSoFar = 0;
 for (var i = 1; i < vector.length; i++) {
  for (var j = 1; j < vector.length; j++) {
   var sum = 0;
   for (var n = i; n <= j; n++) {
    sum += vector[n];
   }
   if (sum > maxSoFar) { maxSoFar = sum; }
  }
 }
 return maxSoFar
}
```

Position: Ln 9, Ch 41 Total: Ln 13, Ch 265

[✓] Toggle editor

Get feedback

Programming and style errors

Error:

```
Problem at line 14 character 17: Missing semicolon.
   return maxSoFar
```

Functionality

- **Array with positive values only (1, 1, 2)**
  1. function named subSum exists
  2. failed, expected: 6 result: 5
- **Array with negative values only (0, 2, 2)**
  1. function named subSum exists
  2. With array of negative values only, maximum subsum is zero.: 0
- **Example given in the exercise description (0, 2, 2)**
  1. function named subSum exists
  2. okay: 187
- **Typical corner cases of the input length (1, 2, 3)**
  1. function named subSum exists
  2. With empty array, maximum subsum is zero.: 0
  3. Array with olement only., expected: 6 result: 0

Complexity metrics

Your algorithm looks quite complex, perhaps O(N^3)

Figure 5.3. Assignment description (top left corner), code editor for solving the assignment (lover left corner) and feedback produced on client side (on right) (figure originally published in Publication VIII).

tially extendable to some other programming languages with little effort. Python, for example, has an interpreter written in JavaScript and can thus be executed in most browsers. Low level features and libraries like network access are not supported because a browser does not have access to all the resources. This problem can be avoided either when designing the assignments or by mocking out the libraries.

5.3.3 HTML and JavaScript in Feedback

Client-side assessment removes the need for sandboxing the execution because unknown code is no longer executed in an environment where it could harm others. Despite this improved security, client-side assessment introduces a new problem of how to trust the results originating from a client. It may be easier to submit fake results than do the actual assignment. Moreover, this does not even require special skills, as browser plugins to modify any submitted data already exist.

In Publication IV we did assessment on the server side but the construction of visualizations was on the client side. Submitting fake results is not possible because tests are executed on the server side but at the same time, there is no need to install visualization tools to the server because visualizations are constructed on the client side. Our technical motivation was to add visual feedback to the existing assessment platforms without or with minimal changes to the platforms. To achieve this, we used "textual feedback" containing visualizations expressed in HTML and JavaScript. Web-CAT uses the assert descriptions as feedback, which allows those descriptions (i.e. visualizations) to be dynamically constructed.

As mentioned in Section 4.5, we tried two different approaches to construct the visualizations:

- HTML, CSS, or JavaScript can be used to draw the image entirely in the browser. Figure 1.2 on page 4, for example, is an HTML table styled with CSS.
- Image tag with a dynamically constructed url pointing to an external web service (e.g. Google Chart API[4]) constructing the image can also be used. An example of this is the following feedback:

  ```
  <img src="https://chart.googleapis.com/
       chart?cht=gv&chl=digraph{A->B->C->A}">
  ```

  This[5] is rendered to graph with three nodes and directed edges from node A to B, B to C and C to A.

The benefit of the latter approach is that the same test oracle and the same visualizer can be used in different assignments. What needs to be done are a model solution and a configuration class to define how the object graph is constructed. Then, comparison of the object graphs (model solution and student's solution) and construction of the visualizing HTML are the same between all assignments.

5.4 Conclusions

Scope, integrability and interaction are all important factors that should be considered when selecting a visualization system. Many of the visual-

[4]`http://code.google.com/apis/chart/`

[5]To be precise, the URL can be copied to a browser and it will render correctly but when the URL is in an HTML document (e.g. url of an image tag) it needs to be encoded because of the special characters in it. $' >'$, for example, should be encoded to *%3E;*.

ization tools are server-side software but as demonstrated in Section 5.3.3, visualizations can also be constructed on client side. A benefit of this approach is that the client-side environment remains the same when assignments are moved to a new server environment. Thus, sharing assignments with visual feedback is easier if no new visualization tools are needed on server side when assignment implementation are imported.

The lesson to learn from our JavaScript experiments is that although doing assessment (no only visualizations) on client side helps using the same assignments in many environments, the results may be easier to tamper. Teachers should realize this problem if they accept grades from a client without double checking the assessment on server side. On the other hand, doing assessment also on server side cancels some of the benefits (e.g. mobility) of client-side assessment. However, for self-studying purposes client-side assessment is an ideal solution, which allows assignments to be used even offline.

Although we relied on JavaScript to implement the tools presented earlier, there are other options as well. Adobe Flash, Microsoft Silverlight, and Java applets, for example, are often used to build Rich Internet Applications (RIAs). From many options, we selected JavaScript because it works out-of-the-box in most environments and we subjectively believed it has the best chances to survive the battle of RIA technologies.

6. Mutation Analysis vs. Structural Coverage as Feedback

The problem of *how to better evaluate students' testing skills* was raised in Chapter 1. Automated evaluation of students' tests is often based on structural coverage, that is what parts of the program under test are visited by the test. This, however, can tell very little about the ability of the tests to find defects. For example,

- `assertTrue(1 < 2); fibonacci(6);`
- `assertTrue(fibonacci(6) >= 0);`
- `assertEquals(8,fibonacci(6));`

all achieve the same structural coverage, although their ability to tell how well the fibonacci method under test works is quite different.

Mutation analysis provides an alternative metric to measure test adequacy. It is a well-known technique performed on a set of unit tests by seeding simple programming errors into the program code to be tested. Each combination of errors applied to the code creates what is called a *mutant*. These "mutants" are generated systematically in large quantities and the examined test suite is run on each of them. The theory is that the test suite that detects more generated defective programs is better than the one that detects fewer [24]. My research question related to using mutation analysis to provide feedback on students' tests is:

> What are the possible strengths and weaknesses of mutation analysis when compared to code coverage-based metrics when both are used to give feedback on students' testing skills?

The research method we applied to answer this question in Publication IX, is to compare the structural coverage of the tests submitted by students to the *mutation coverage*[1] of the same submissions and investi-

[1] Percentage of mutants detected.

gate the feedback manually to find out the pros and cons of it. Mutation scores were calculated by using a tool called Javalanche [83].

Comparison of the Metrics

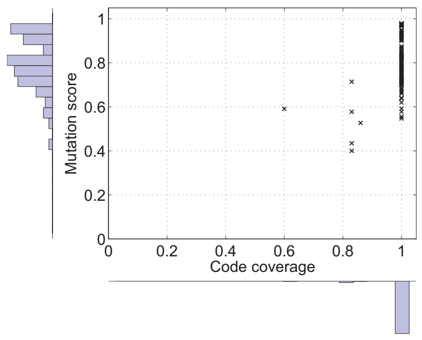
Figure 6.1 illustrates the relationship between mutation score and structural test coverage in three assignments we used in our study. Histograms on each axis show the distribution of the respective variables. Most students get full points from the traditional metric, but the mutation scores vary a lot. This suggests that despite good structural coverage, tests written by students are actually not efficient in finding defects from their programs.

A well known problem in mutation analysis is that some mutants do not behave differently when compared to the original program from which they are derived. Such mutants are called equivalent mutants. If mutation coverage[2] is calculated automatically, the portion of equivalent mutants distorts the results. A significant property in JavaLanche, when compared to many preceding mutation analysis tools, is the notably low number of equivalent mutants. However, to verify that equivalent mutants do not cause the results presented in Figure 6.1, we also examined some submissions manually. Figure 6.1 presents data collected from students' submissions to three different assignments. In the first and second assignment, bad mutation score predicted low quality of tests. Submissions to the third assignment, however, produced many equivalent mutants and poor mutation score did not directly indicate poor test quality.

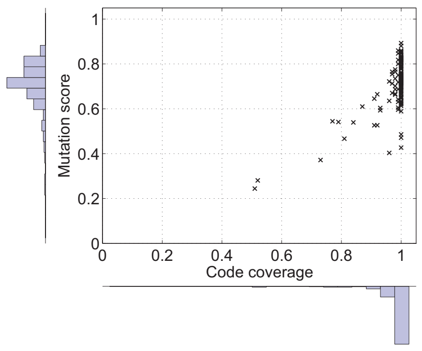
Manually created faulty programs would not suffer from equivalent mutants but they also could not assess tests of functions not defined in the assignment description. Mutation analysis is able to give feedback also from additional helper methods created and tested by students.

We concluded that mutation analysis can reveal tests that are created to fool the assessment system but the suitability for contributing to the grade depends on the assignments. It is possible that typical solutions to some problems contain structures that JavaLanche fails to mutate well (i.e. many equivalent mutants are produced). While the information from assignments that fit poorly for mutation analysis is most easily interpreted and used by a teacher, the results could be valuable to the students as well.

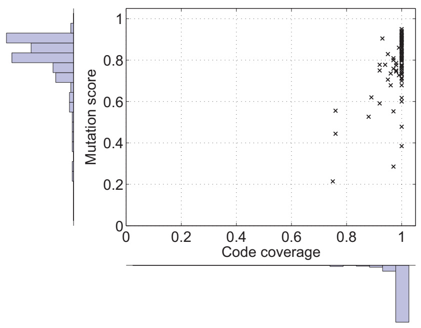
[2]The portion of mutants detected.



(a) Assignment 1



(b) Assignment 2



(c) Assignment 3

Figure 6.1. Scatter plots of code coverages and mutation scores of the assignments analyzed in Publication IX.

One of the problems, pointed out e.g. by Greening [37], is that an automatically assessed programming assignment is essentially one of reproduction. I believe mutation analysis can address this problem. It may allow intentionally not well specified assignments where students define how their programs should behave and test it. Mutation analysis, measuring the quality of tests, can be applied only if the tests pass. The fact that tests test something that can fail can be verified with mutation analysis. This, however, still does not provide enough information about the meaningfulness of requirements selected by a student. Future research is needed to find the balance how much freedom students should be allowed when mutation analysis is used to grade their performance.

7. Conclusions

The main objective of this thesis has been to improve the automated assessment of programming assignments from the perspective of tool developers. To achieve this, I have explored three research questions:

Q1 How to provide visual feedback on automatically assessed programming assignments?

Q2 How to allow assignment implementations and visual feedback to be more independent from assessment platforms?

Q3 How to better evaluate students' testing skills?

Let us pause for bit to reflect on these questions and put them on a broader context. The users of assessment tools are teachers and students. Assessment tools are only one of the many factors affecting to the actions of all these users. Some of the other factors and the role of assessment tools in the process of education are illustrated in Figure 7.1. The figure is adapted from Engeström's activity system diagram [32] – two activity systems. The right side of the figure presents the student's view for learning programming, the the teacher's is on the left. The three topmost items in an activity systems diagram are always the subject (i.e., teacher or student), the mediating tools (i.e., automated assessment tools), and the object (i.e., teach programming or learn programming). These have been all discussed in this thesis. Teachers' and students' connection to teaching and learning has also been discussed. This has happened mainly through the assessment tools. In addition, I have touched upon the connection between assessment tools and the teacher community. This connection is key to sharing programming assignments and making assignments easier to share. These topics of my interest are highlighted in Figure 7.1. The remaining corners of the activity system diagrams illustrate the formal and informal rules related to the learning environment and teachers' or learners' way to share or divide their work. These are all out of the scope of this thesis.

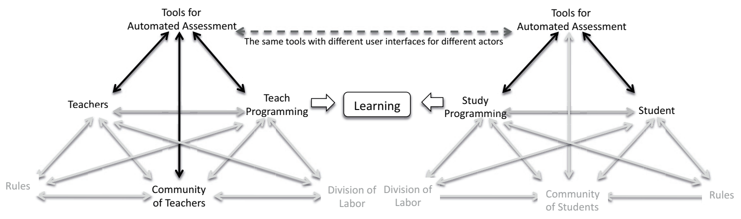


Figure 7.1. Two activity systems both describing learning programming with assessment tools. The right side of the figure presents the activity systems related to students and the system on left relates to teachers. Areas of interest in this thesis are drawn with black (cf. grey) color.

The rest of this final chapter is divided into four sections. Section 7.1 concludes Q1 and the visual feedback part of Q2. Section 7.2 answers the rest of Q2 and Section 7.3 summarizes and concludes the findings related to Q3. Finally, future research problems and predictions related to the future of automated assessment of programming assignments are presented in Section 7.4.

7.1 Visual Feedback and Portability

The existing literature suggests that the use of engaging software visualizations can provide a positive impact on learning. However, visualizations are rarely, if at all, used as part of automated feedback on the functionality of students' source code (see Chapters 2 and 3). This motivates the question of *how to provide visual feedback from automatically assessed programming assignments* (see Chapter 4).

It appears that controlling the level of detail in visual feedback and especially raising the level of abstraction are important. Approaches to achieve this have been discussed. Section 4.4 describes how a test input generation method called *generalized symbolic execution with lazy initialization*, originally presented by Khursid et al. [56], can be applied to construct visualizations of test data on a high abstraction level.

Visual feedback should be implemented in a way that is easy to port from one assessment platform to others. This is the topic of Sections 5.1 and 5.3.3. The first approach suggested there is to integrate a software visualization tools to the assessment platform. In this case, one should pay attention to the capabilities and requirements related to the intended use of visualizations. This includes a number of considerations, beginning with the question of what should be visualized. Visualization–learner interaction and the interaction between visualization tools and the assessment platform should also be designed. Finally, one should decide how visualizations are embedded in the feedback. The other approach, explored in more depth, originated from the observation that most assessment platforms are web based. This creates a possibility for textual feedback to include JavaScript and HTML. Thus, visualizations and even interactive elements can be delivered from the assessment platform to the browsers of students in a way that requires only minimal, if any, changes to the platform. Details of the latter approach have been discussed in Section 5.3.3 and examples of visual feedback produced with it have been provided in Figures 4.3 on page 37 and 1.2 on page 4.

In Section 4.5 different kinds of automated feedback were compared using various performance metrics. Although students were not trained to deal with the visual feedback, no performance differences between the groups were found. This is an interesting result because visual feedback based on automatically extracted object graphs can take less time to prepare than textual feedback of good quality. It has been pointed out that teaching how to interpret and apply visualizations can be extremely important for learning [11]. Therefore, it would be interesting to see how this kind of support would affect to the results in our study.

In conclusion, there are no major technical obstacles that prevent teachers from implementing visual feedback in automatically assessed programming assignments. Further research to evaluate the effectiveness of visual feedback on programming assignments is still needed.

7.2 Portability of Assignments

Assignment implementations are often difficult to port from one assessment platform to another. The two approaches making assignment implementations easier to share and discussed in my thesis are:

- Use of learning management systems, such as Moodle, as an assignment-sharing platform. Unfortunately it turned out that despite clear demand, mainstream learning management systems in general, and

Moodle in particular, do not yet provide adequate support for automatic code management, assessment, and visualization.

- Assessment on the client side instead of on a server. Although this enables reuse of assignments and removes the need for sandboxing the execution, the results are easier to tamper with. This is a problem if assignments are used for grading purposes but not if assignments are self-study material. In addition, client-side assessment has the potential to ease plugin development for learning management systems.

The idea of client-side assessment has been presented by others as well. For example, use of tailored email clients to assess programming assignments client side was suggested in 2009 by Sant [82]. Although I have focused on web browsers, the underlying idea is similar. In addition, client-side assessment has connections to certain 2nd generation assessment platforms that executed programs with students permissions in a shared Unix environment. For example, at Helsinki University of Technology we applied this in mid-1990s. Students used the assessment tools from command line and tests were executed by using the student's unix account. At the end of the assessment, the grade was stored in a text file. Because assessment was executed with the student's privileges, all students had write permissions to the grade files. This created an easy opportunity for each student to overwrite any of the results if they knew where the results were stored. This problem is very similar to the reliability problem of in-browser assessment.

Other approaches that could improve assignment mobility include a common assignment format, hosting assignments, use of Sharable Content Object Reference Model (SCORM). These have not been discussed in this thesis.

7.3 Assessment of Testing Skills

The motivation for asking *how to better evaluate students' testing skills* and seeking alternative metrics for measuring the quality of students' tests is that grading based on structural coverage only can lead students to adopt poor testing practices. Based on our experiences, students may end up writing tests with assertions of poor quality if the ability of the tests to detect bugs is not evaluated.

Mutation analysis is a well-known technique performed on a set of unit tests by seeding simple programming errors into the program code to be tested. Each combination of errors applied to the code creates what is called a *mutant*. These ”mutants” are generated systematically in large quantities and the examined test suite is run on each of them. The theory is that the test suite that detects more mutants is better than the one that detects less. So far, this technique has not been commonly applied in automated assessment.

The pros and cons of mutation analysis in automated assessment are discussed in Chapter 6. A clear benefit of applying mutation analysis in assessment is that it can reveal tests that were created to fool the assessment system. However, the results can be valuable for learners as well as they can reveal what kinds of faulty programs the tests are not able to detect. However, applying mutation analysis for grading is problematic because some of the mutants may actually be functionally identical with the original program. Use of predefined faulty programs instead of mutants can provide more accurate feedback but only on aspects that are fixed in the assignment description.

7.4 Future Directions

One problem of client-side assessment, that needs to be addressed in the future, is that the results are too easy to fake. To solve this, it may be possible to execute the program on the client side and submit the output elsewhere for comparison. In this case, securing the assessment server is easier. A student’s program can harm only the student’s own environment and the expected output is not known in the environment where the program is executed. The downside of this approach is that the assessment is not totally self-standing. More research is needed to better understand the benefits and challenges of client-side assessment.

Another avenue for future work is to have many small independent web services performing dedicated tasks that contribute to automated assessment. Calling Google chart API from the client, as we did in Publication IV, is an example of this. Something similar could also be implemented within the field of algorithm visualization, as we have also proposed [52]. The existence of such services could help in integrating programming assignments in learning management systems.

Automatically assessed programming assignments have been argued not to foster creativity because students merely reproduce something that the teacher has already implemented. Mutation analysis can address this problem. It may be possible to design programming assignments where everything is not strictly specified. Instead, the learner needs to make decisions regarding how his or her program should work and to test this behavior. The quality of the solution is then indirectly estimated through measuring the quality of the passing tests with mutation analysis. This, although checking that the selected requirements are not trivial, still does not provide enough information about the meaningfulness of the selections made by the student. Future research is needed to find a balance regarding the freedom students are allowed.

I believe that online learning materials will grow in importance in the future and that the nature of such material will become increasingly visual and interactive. Assessing programming assignments and constructing visual feedback client side may play an important role in this development.

Bibliography

[1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Koli Calling '06: Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 141–142. ACM, New York, NY, USA, 2006. doi:10.1145/1315803.1315831.

[2] Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy web-based animation from CS 0 to advanced CS courses. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 162–166. ACM, New York, NY, USA, 2003. doi:10.1145/611892.611959.

[3] Kirsti Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005. doi:10.1080/08993400500150747.

[4] Kirsti Ala-Mutka and Hannu-Matti Järvinen. Assessment process for programming assignments. In *ICALT '04: 4th IEEE International Conference on Advanced Learning Technologies*, pages 181–185. IEEE Computer Society, Washington, DC, USA, 2004. doi:10.1109/ICALT.2004.1357399.

[5] Anton Alstes and Janne Lindqvist. VERKKOKE: learning routing and network programming online. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 91–95. ACM, New York, NY, USA, 2007. doi:10.1145/1268784.1268813.

[6] Mario Amelung, Peter Forbrig, and Dietmar Rösner. Towards generic and flexible web services for e-assessment. In *ITiCSE '08: Proceedings of the 13th annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 219–224. ACM, New York, NY, USA, 2008. doi:10.1145/1384271.1384330.

[7] Mario Amelung, Michael Piotrowski, and Dietmar Rösner. Educomponents: experiences in e-assessment in computer science education. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 88–92. ACM, New York, NY, USA, 2006. doi:10.1145/1140124.1140150.

[8] Lorin W. Anderson, David R. Krathwohl, Peter W. Airasian, Kathleen A. Cruikshank, Richard E. Mayer, Paul R. Pintrich, James Raths, and Merlin C. Wittrock. *A taxonomy for learning, teaching, and assessing: A revi-*

sion of Bloom's taxonomy of educational objectives. Longman, New York, NY, USA, 2001.

[9] Mordechai Ben-Ari. Constructivism in computer science education. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 257–261. ACM, New York, NY, USA, 1998. doi:10.1145/273133.274308.

[10] Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[11] Ronit Ben-Bassat Leny, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40:1–15, 2003. doi:10.1016/S0360-1315(02)00076-3.

[12] Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and Abdullah Modh Zin. Ceilidh as a course management support system. *Journal of Educational Technology Systems*, 22(3):235–250, 1993.

[13] John Biggs and Kevin F. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, 1982.

[14] John Biggs and Catherine Tang. *Teaching for Quality Learning at University : What the Student Does (3rd Edition)*. Open University Press, 2007.

[15] Benjamin S. Bloom. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley, 1956.

[16] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007. doi:http://dx.doi.org/10.1016/j.jss.2006.07.009.

[17] Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. PILOT: an interactive tool for learning and grading. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 139–143. ACM, New York, NY, USA, 2000. doi:10.1145/330908.331843.

[18] Neil Brown, Phil Stevens, and Michael Kölling. Greenroom: a teacher community for collaborative resource development. In *ITiCSE '10: Proceedings of the fifteenth annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 305–305. ACM, New York, NY, USA, 2010. doi:10.1145/1822090.1822181.

[19] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. How shall we assess this? *SIGCSE Bull.*, 35:107–123, 2003. doi:10.1145/960492.960539.

[20] Michael E. Caspersen and Jens Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 111–122. ACM, New York, NY, USA, 2007. doi:10.1145/1288580.1288595.

[21] Ming-Yu Chen, Jyh-Da Wei, Jeng-Hung Huang, and D. T. Lee. Design and applications of an algorithm benchmark system in a computational problem solving environment. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 123–127. ACM, New York, NY, USA, 2006. doi:10.1145/1140124.1140159.

[22] Frank Coffield, David Moseley, Elaine Hall, and Kathryn Ecclestone. Should we be using learning styles? what research has to say in practice – report of the learning and skills. Technical report, Development Agency, Regent Arcade House, Argyle St. London, UK, 2004.

[23] Joel Coffman and Alfred C. Weaver. Electronic commerce virtual laboratory. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, pages 92–96. ACM, New York, NY, USA, 2010. doi:10.1145/1734263.1734295.

[24] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, 1978. doi:10.1109/C-M.1978.218136.

[25] Françoise Détienne. Expert programming knowledge: A schema-based approach. In J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore (editors), *Psychology of Programming*, pages 205–222. Academic Press, London, 1990.

[26] Stephan Diehl (editor). *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science (LNCS)*. Springer Verlag, Dagstuhl, Germany, 2002.

[27] Jing Dong, Yongtao Sun, and Yajing Zhao. Design pattern detection by template matching. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 765–769. ACM, New York, NY, USA, 2008. doi:10.1145/1363686.1363864.

[28] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):Article 4 (13 pages), 2005. doi:10.1145/1163405.1163409.

[29] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)*, 3(3):Article 1 (24 pages), 2003. doi:10.1145/1029994.1029995.

[30] Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155. ACM, New York, NY, USA, 2003. doi:10.1145/949344.949390.

[31] Stephen H. Edwards, Jürgen Börstler, Lillian N. Cassel, Mark S. Hall, and Joseph Hollingsworth. Developing a common format for sharing programming assignments. *SIGCSE Bull.*, 40:167–182, 2008. doi:10.1145/1473195.1473240.

[32] Yrjö Engeström. *Learning by expanding: an activity-theoretical approach to developmental research*. Oriente-Konsultit, Helsinki, Finland, 1987.

[33] R. M. Felder and L. K. Silverman. Learning styles and teaching styles in engineering education. *Engineering Education*, 78(7):674–681, 1988.

[34] Michal Forišek. Security of Programming Contest Systems. In Valentina Dagiene and Roland Mittermeir (editors), *Information Technologies at School*, pages 553–563. Institute of Mathematics and Informatics, Vilnius, Lithuania, 2006.

[35] Manuel Freire. Visualizing program similarity in the ac plagiarism detection system. In *AVI '08: Proceedings of the working conference on Advanced visual interfaces*, pages 404–407. ACM, New York, NY, USA, 2008. doi:10.1145/1385569.1385644.

[36] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. Developing a computer science-specific learning taxonomy. *SIGCSE Bull.*, 39(4):152–170, 2007. doi:10.1145/1345375.1345438.

[37] Tony Greening. Emerging constructivist forces in computer science education: Shaping a new future. In Tony Greening (editor), *Computer science education in the 21st century*, pages 47–80. Springer Verlag, 1999.

[38] Eladio Gutiérrez, María A. Trenas, Julián Ramos, Francisco Corbera, and Sergio Romero. A new Moodle module supporting automatic verification of VHDL-based assignments. *Computers & Education*, 54:562–577, 2010. doi:10.1016/j.compedu.2009.09.006.

[39] John Hamer. A lightweight visualizer for Java. In *Proceedings of Third Program Visualization Workshop*, Research Report CS-RR-407, pages 54–61. University of Warwick, Warwick, UK, 2004.

[40] Michael T. Helmick. Interface-based programming assignments and automatic grading of Java programs. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 63–67. ACM, New York, NY, USA, 2007. doi:10.1145/1268784.1268805.

[41] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12:272–275, 1969. doi:10.1145/362946.362981.

[42] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003. doi:10.1023/A:1026364126982.

[43] Winfred F. Hill. *Learning: A Survey of Psychological Interpretations*. Allyn et. Bacon, 7th edition, 2001.

[44] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3:528–529, 1960. doi:10.1145/367415.367422.

[45] 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. doi:10.1006/jvlc.2002.0237.

[46] David Jackson and Michelle Usher. Grading student programs using ASSYST. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339. ACM, New York, NY, USA, 1997. doi:10.1145/268084.268210.

[47] Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: Future of Software Engineering*, pages 199–213. IEEE Computer Society, Washington, DC, USA, 2007. doi:10.1109/FOSE.2007.26.

[48] Daniel Jimenez-Gonzalez, Carlos Alvarez, David Lopez, Joan-M. Parcerisa, Javier Alonso, Christian Perez, Ruben Tous, Pere Barlet, Montse Fernandez, and Jordi Tubella. Work in progress – improving feedback using an automatic assessment tool. In *FIE 2008: 38th annual Frontiers in Education Conference*, pages S3B–9 – T1A–10. 2008. doi:10.1109/FIE.2008.4720591.

[49] Colin G. Johnson and Ursula Fuller. Is Bloom's taxonomy appropriate for computer science? In Anders Berglund and Mattias Wiggberg (editors), *Koli Calling '06: Proceedings of the 6th Baltic Sea conference on Computing Education Research*, pages 120–123. ACM, New York, NY, USA, 2006. doi:10.1145/1315803.1315825.

[50] Mike Joy, Nathan Griffiths, and Russell Boyatt. The BOSS online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)*, 5:Article 2 (28 pages), 2005. doi:10.1145/1163405.1163407.

[51] Mike Joy and Michael Luck. Effective electronic marking for on-line assessment. *SIGCSE Bull.*, 30:134–138, 1998. doi:10.1145/290320.283096.

[52] Ville Karavirta and Petri Ihantola. Initial set of services for algorithm visualization. In *Proceedings of the 6th Program Visualization Workshop*, pages 67–71. Darmstadt, Germany, 2011.

[53] Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro – A tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33. Department of Computer Science, University of Warwick, UK, 2004.

[54] Ville Karavirta, Ari Korhonen, Jussi Nikander, and Petri Tenhunen. Effortless creation of algorithm visualization. In *Koli Calling '02: Proceedings of the Second Annual Finnish / Baltic Sea Conference on Computer Science Education*, Report Series A 1, pages 52–56. University of Joensuu, Department of Computer Science, Joensuu, Finland, 2002.

[55] Ville Karavirta, Ari Korhonen, and Petri Tenhunen. Survey of effortlessness in algorithm visualization systems. In *Proceedings of the Third Program Visualization Workshop*, pages 141–148. Department of Computer Science, University of Warwick, UK, 2004.

[56] Sarfraz Khursid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings 9th International Conference on Tools and Algorithms for Construction and Analysis*, volume 2619 of *Lecture Notes in Computer Science (LNCS)*, pages 553–568. Springer-Verlag, 2003.

[57] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi:10.1145/360248.360252.

[58] David A. Kolb (editor). *Experiential Learning: Experience as the Source of Learning and Development*. Prentice-Hall Inc, New Jersey, USA, 1984.

[59] Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Koli Calling '03: Proceedings Third Annual Baltic Conference on Computer Science Education*, pages 48–56. Department of Computer Science, Report B-2003-3, University of Helsinki, Helsinki University Printing House, 2003.

[60] Lucas Layman, Travis Cornwell, and Laurie Williams. Personality types, learning styles, and an agile approach to software engineering education. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 428–432. ACM, New York, NY, USA, 2006. doi:10.1145/1121341.1121474.

[61] Yingli Liang, Quanbo Liu, Jun Xu, and Dongqing Wang. The recent development of automated programming assessment. In *CiSE '09: Proceedings of International Conference on Computational Intelligence and Software Engineering*, pages 1–5. IEEE Computer Society, Washington, DC, USA, 2009. doi:10.1109/CISE.2009.5365307.

[62] Raymond Lister and John Leaney. Introductory programming, criterion-referencing, and bloom. *SIGCSE Bull.*, 35:143–147, 2003. doi:10.1145/792548.611954.

[63] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. Investigating the viability of mental models held by novice programmers. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 499–503. ACM, New York, NY, USA, 2007. doi:10.1145/1227310.1227481.

[64] Lauri Malmi, Ville Karavirta, Ari Korhonen, and Jussi Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal of Educational Resources in Computing (JERIC)*, 5(3):Article 7 (23 pages), 2005. doi:10.1145/1163405.1163412.

[65] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. Learning computer science concepts with scratch. In *ICER '10: Proceedings of the Sixth international workshop on Computing education research*, pages 69–76. ACM, New York, NY, USA, 2010. doi:10.1145/1839594.1839607.

[66] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for gui testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267. ACM, New York, NY, USA, 2001. doi:10.1145/503209.503244.

[67] Antonija Mitrovic and Stellan Ohlsson. Evaluation of a constraint-based tutor for a database language. *International Journal of Artificial Intelligence in Education*, 10:238–256, 1999.

[68] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 373–376. ACM, New York, NY, USA, 2004. doi:10.1145/989863.989928.

[69] Isabel Briggs Myers, Mary H. McCaulley, Naomi L. Quenk, and Allen L. Hammer. *MBTI Manual (A guide to the development and use of the Myers Briggs type indicator)*. Consulting Psychologists Press, 3rd ed. edition, 1998.

[70] Niko Myller, Roman Bednarik, Erkki Sutinen, and Mordechai Ben-Ari. Extending the engagement taxonomy: Software visualization and collaborative learning. *Trans. Comput. Educ.*, 9:7:1–7:27, 2009. doi:10.1145/1513593.1513600.

[71] Thomas L. Naps, James R. Eagan, and Laura L. Norton. Jhave – an environment to actively engage students in web-based algorithm visualizations. *SIGCSE Bull.*, 32:109–113, 2000. doi:10.1145/331795.331829.

[72] 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. *SIGCSE Bull.*, 35(2):131–152, 2003.

[73] Kevin A. Naudé, Jean H. Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, 2010. doi:10.1016/j.compedu.2009.09.005.

[74] Pete Nordquist. Providing accurate and timely feedback by automatically grading student programming labs. *J. Comput. Small Coll.*, 23(2):16–23, 2007.

[75] Dale Parsons and Patricia Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *ACE '06: Proceedings of the 8th Austalian conference on Computing education*, pages 157–163. Australian Computer Society, Inc., Darlinghurst, Australia, 2006.

[76] Harold Pashler, Mark McDaniel, Doug Rohrer, and Robert Bjork. Learning styles: Concepts and evidence. *Psychological Science in the Public Interest*, 9(3):105–119, 2008. doi:10.1111/j.1539-6053.2009.01038.x.

[77] Arnold Pears, Stephen Seidman, Crystal Eney, Päivi Kinnunen, and Lauri Malmi. Constructing a core literature for computing education research. *SIGCSE Bull.*, 37(4):152–161, 2005. doi:10.1145/1113847.1113893.

[78] 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.

[79] Kenneth A. Reek. The TRY system -or- how to avoid testing student programs. *SIGCSE Bull.*, 21:112–116, 1989. doi:10.1145/65294.71198.

[80] Guido Rößling and Bernd Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

[81] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *ITiCSE 01: Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 133–136. ACM Press, New York, Canterbury, UK, 2001.

[82] Joseph A. Sant. "Mailing it in": email-centric automated assessment. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 308–312. ACM, New York, NY, USA, 2009. doi:10.1145/1562877.1562971.

[83] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for Java. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineeringon European software engineering conference and foundations of software engineering symposium*, pages 297–298. ACM, New York, NY, USA, 2009. doi:10.1145/1595696.1595750.

[84] Wilfrid Sellars. Philosophy and the scientific image of a man. In Robert G. Colodny (editor), *Frontiers of Science and Philosophy*, pages 35–78. University of Pittsburgh Press, Pittsburgh, 1962.

[85] Judy Sheard, S. Simon, Margaret Hamilton, and Jan Lönnberg. Analysis of research into the teaching and learning of programming. In *ICER '09: Proceedings of the fifth international workshop on Computing education research workshop*, pages 93–104. ACM, New York, NY, USA, 2009. doi:10.1145/1584322.1584334.

[86] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. In *Readings in artificial intelligence and software engineering*, pages 507–521. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[87] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with Marmoset: designing and using an advanced submission and testing system for programming courses. In *ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 13–17. ACM, New York, NY, USA, 2006. doi:10.1145/1140124.1140131.

[88] Christopher W. Starr, Bill Manaris, and RoxAnn H. Stalvey. Bloom's taxonomy revisited: specifying assessable learning objectives in computer science. *SIGCSE Bull.*, 40(1):261–265, 2008. doi:10.1145/1352322.1352227.

[89] Hussein Suleman. Automatic marking with Sakai. In *SAICSIT '08: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, pages 229–236. ACM, New York, NY, USA, 2008. doi:http://doi.acm.org/10.1145/1456659.1456686.

[90] Mate' Sztipanovits, Kai Qian, and Xiang Fu. The automated web application testing (AWAT) system. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference*, pages 88–93. ACM, New York, NY, USA, 2008. doi:10.1145/1593105.1593128.

[91] Ahmad Taherkhani, Lauri Malmi, and Ari Korhonen. Algorithm recognition by static analysis and its application in students' submissions assessment. In *Koli Calling '08: Proceedings of the 8th Koli Calling International Conference on Computing Education Research*, pages 88–91. ACM, New York, NY, USA, 2009. doi:10.1145/1595356.1595372.

[92] Ahmad Taherkhani, Lauri Malmi, and Ari Korhonen. Using roles of variables in algorithm recognition. In *Koli Calling '09: Proceedings of the 9th Koli Calling International Conference on Computing Education Research*, pages 1–10. Uppsala University, Uppsala, Sweden, 2010.

[93] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform. In *SEAA '08: 34th Euromicro Conference on Software Engineering and Advanced Applications*, pages 293 –302. IEEE Computer Society, Washington, DC, USA, 2008. doi:10.1109/SEAA.2008.17.

[94] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *ACE '04: Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, pages 317–325. Australian Computer Society, Inc., Darlinghurst, Australia, 2004.

[95] David W. Valentine. CS educational research: a meta-analysis of SIGCSE technical symposium proceedings. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 255–259. ACM, New York, NY, USA, 2004. doi:10.1145/971300.971391.

[96] Esa Vihtonen and Eugene Ageenko. Viope-computer supported environment for learning programming languages. In *TICE2002: The Proceedings of International Symposium on Technologies of Information and Communication in Education for Engineering and Industry*, pages 371–372. Lyon, France, 2002.

[97] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM, New York, NY, USA, 2004. doi:10.1145/1007512.1007526.

[98] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107, 2007. doi:10.1016/j.infsof.2006.03.001.

[99] Carol Zander, Lynda Thomas, Beth Simon, Laurie Murphy, Renée McCauley, Brian Hanks, and Sue Fitzgerald. Learning styles: novices decide. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 223–227. ACM, New York, NY, USA, 2009. doi:10.1145/1562877.1562948.

[100] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. doi:10.1145/267580.267590.

Errata

Program 4 on page 88 of Publication III is wrong and does not match with Figure 2 (page 87 of Publication III). Line 7 of Program 4 should be: `static {v.add(null);}` in which case constructor should add *this* to the vector v whenever a new object is created. In addition, *next* labels in Figure 2 should state *right* and getNext on page 88 should be getRight.

Aalto-DD 131/2011

ISBN 978-952-60-4397-5
ISBN 978-952-60-4398-2 (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