TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tuukka Ahoniemi

Efficient Use of Teaching Technologies with Programming Education

Julkaisu 1301 • Publication 1301

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1301
Tampere University of Technology. Publication 1301

Tuukka Ahoniemi

Efficient Use of Teaching Technologies with Programming Education

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB222, at Tampere University of Technology, on the 28th of May 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3526-0 (printed)
ISBN 978-952-15-3535-2 (PDF)
ISSN 1459-2045

ABSTRACT

Learning and teaching programming are challenging tasks that can be facilitated by using different teaching technologies. Visualization systems are software systems that can be used to help students in forming proper mental models of executed program code. They provide different visual and textual cues that help student in abstracting the meaning of a program code or an algorithm. Students also need to constantly practice the skill of programming by implementing programming assignments. These can be automatically assessed by other computer programs but parts of the evaluation need to be assessed manually by teachers or teaching assistants.

There are a lot of existing tools that provide partial solutions to the practical problems of programming courses: visualizing program code, assessing student programming submissions automatically or rubrics that help keeping manual assessment consistent. Taking these tools into use is not straightforward. To succeed, the teacher needs to find the suitable tools and properly integrate them into the course infrastructure supporting the whole learning process. As many programming courses are mass courses, it is a constant struggle between providing sufficient personal guidance and feedback while retaining a reasonable workload for the teacher.

This work answers to the question *"How can the teaching of programming be effectively assisted using teaching technologies?"* As a solution, different learning taxonomies are presented from Computer Science perspective and applied to visualization examples so the examples could be used to better support deeper knowledge and the whole learning process within a programming course. Then, different parts of the assessment process of programming assignments are studied to find the best practices in supporting the process, especially when multiple graders are being used, to maintain objectivity, consistency and reasonable workload in the grading.

The results of the work show that teaching technologies can be a valuable aid for the teacher to support the learning process of the students and to help in the practical organization of the course without hindering the learning results or personalized feedback the students receive from their assignments. This thesis presents new visualization categories that allow deeper cognitive development and examples on how to integrate them efficiently into the course infrastructure. This thesis also presents a survey of computer-assisted assessment tools and assessable features for teachers to use in their programming assignments. Finally, the concept of rubric-based assessment tools is introduced to facilitate the manual assessment part of programming assignments.

PREFACE

My intention and interest around Computer Science Education Research has always been very practical: to create tools and report findings that can be directly adapted into the classroom. With this thesis I'm summarizing my ten years of work around solving practical issues I have faced when teaching or training people to write computer programs. I have been very fortunate to work with something I really enjoy and even more fortunate by having the greatest people around me.

During these years, many people have helped, influenced and inspired me. Firstly, I would like to express my deepest gratitude to my always-positive supervisor, professor Hannu-Matti Järvinen, who has been guiding and mentoring my research for the past ten years and especially for encouraging and helping me to finish my work towards the thesis. I would also like to thank professor Tommi Mikkonen for kindly guiding me back to unfinished research, providing me the practical opportunity for finishing the thesis and giving valuable feedback during the writing process.

I would like to thank professor Mordechai Ben-Ari and professor Tapio Salakoski for being the pre-examiners of the thesis and providing valuable observations and feedback for the work.

I want to thank Essi Isohanni (Lahtinen) for having the courage to hire myself, an enthusiast CS student, as a teaching assistant and introducing me to the world of CSER. This then quite soon sparked into a magnificent pirate adventure of several joint research projects, eventually resulting into this thesis. Also, thank you for the feedback and suggestions during the writing process of the thesis!

Over the years, I have appreciated being part of the Computer Science Research community and huge part of this work is done jointly with my colleagues. Especially I would like to thank Ville Karavirta, Petri Ihantola, Otto Seppälä, Juha Sorva and Kirsti Ala-Mutka. It has been a pleasure working with you. Also, a big thanks to Veli-Pekka Eloranta for being an inspiring peer researcher and a great friend!

For the past years, I have worked outside the academia and a special recognition goes to *The Man Himself*, Tino Pyssysalo who has enabled and encouraged me to continue my research along our daily work in Digia.

I also want to thank my parents for encouraging me towards academic research. Thank you to all my family, in-laws and friends for all the support and cheers. Finally, and most importantly, I want to thank my dearest Maria, Markus, Tuomas and Rasmus: *Thank you for being so awesome and being there for me!*

Tampere, April 2015,

Tuukka Ahoniemi

CONTENTS

1. Introduction . . . 1
  1.1 Thesis within Computer Science Education Research . . . 3
  1.2 Research Questions . . . 5
  1.3 Organization of the Thesis . . . 7
2. Introduction to Visualization Systems . . . 9
  2.1 What Are Visualizations? . . . 9
  2.2 Algorithm Visualizations . . . 11
  2.3 Program Visualizations . . . 13
  2.4 Usage and Effectiveness of Visualizations in Education . . . 19
    2.4.1 Early Studies on AV Effectiveness . . . 19
    2.4.2 Integration to Course Infrastructure . . . 19
    2.4.3 Effects in Cognitive Process . . . 20
    2.4.4 How Visualizations Are Used . . . 22
    2.4.5 Teachers' Attitudes towards Visualizations . . . 22
  2.5 Summary . . . 23
3. Towards Deeper Cognitive Levels with Visualizations . . . 25
  3.1 Engagement Taxonomy . . . 25
  3.2 Bloom's Taxonomy of Cognitive Development in CS Education . . . 28
  3.3 Summary . . . 31
4. Tools-Assisted Assessment in Programming Courses . . . 33
  4.1 Assessment and Programming . . . 33
  4.2 Computer-Assisted Assessment . . . 36
  4.3 Semi-Automatic Assessment . . . 38
  4.4 Rubric-Based Assessment Tools . . . 40
  4.5 Summary . . . 42
5. Summary of the Included Publications . . . 45
6. Conclusions . . . 57
  6.1 Efficient Integration of Visualization Systems . . . 57
  6.2 Facilitating the Assessment of Programming Assignments . . . 59
  6.3 Summary of the Results . . . 63
  6.4 Generalization and Limitations of the Work . . . 65
  6.5 Benefits of the Work . . . 66
References . . . 69
7. Publications reprinted . . . 79

LIST OF INCLUDED PUBLICATIONS

(i) Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T., McGee Thompson, D., Riedesel, C., Thompson, E. Developing a Computer Science-Specific Learning Taxonomy *In: Working group reports on ITiCSE on Innovation and technology in computer science education*, June 2007, Dundee, Scotland

(ii) Lahtinen, E., Ahoniemi, T. Visualizations to Support Programming on Different Levels of Cognitive Development *In: The Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 87–94, November 2005.

(iii) Ahoniemi, T., Lahtinen, E. Visualizations in Preparing for Programming Exercise Sessions *In: The Proceedings of The Fourth Program Visualization Workshop*, June 2006, Florence.

(iv) Lahtinen, E., Ahoniemi, T. Kick-Start Activation to Novice Programmers - A Visualization-Based Approach *In: The Proceedings of PVW 2008 - Fifth Program Visualization Workshop*, July 2008 Madrid, Spain.

(v) Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O. Review of Recent Systems for Automatic Assessment of Programming Assignments *In: The Proceedings of 10th Koli Calling International Conference on Computing Education Research*, October 2010, Koli, Finland.

(vi) Ahoniemi, T., Lahtinen, E., Reinikainen, T. Improving Pedagogical Feedback and Objective Grading *In: The Proceedings of SIGCSE'08*, March 2008, Portland, Oregon, USA.

(vii) Ahoniemi, T., Karavirta, V. Analyzing the Use of a Rubric-Based Grading Tool *In: The Proceedings of ITiCSE 2009*, June 2009, Paris, France.

AUTHOR'S CONTRIBUTION TO THE PUBLICATIONS

Publication (i) is an article written by a working group of ITiCSE 2007 conference which the author of this thesis was part of. The author, together with Essi Lahtinen, collected empirical data of taxonomy usage and conducted literature reviews of existing taxonomies prior to the working group meeting. The author, together with Essi Lahtinen and Charles Riedesel, were responsible for defining, designing and writing about the adaptation of Bloom's Taxonomy of Cognitive Development into CS education context: the Matrix Taxonomy, and applying the taxonomy iteratively. The author of this thesis was also part of the finalizing writing process for the whole article.

Publication (ii) presents a theoretical adaptation of Bloom's Taxonomy of Cognitive Development into Program Visualization examples. The original idea for adapting the taxonomy into the practical examples came from Essi Lahtinen. The adaptation to new visualization categories, finding the corresponding features and creating practical examples were done together by the authors. The publication was written jointly by the authors, while Essi Lahtinen was the corresponding author of the publication.

Publication (iii) describes a practical adaptation of the findings of publication (i) in a form of an empirical study around adapting visualizations into programming exercises. The research setup was planned together by the authors and executed in practice by the author of this thesis. The writing process was done jointly while the author of this thesis was the corresponding author of the publication.

Publication (iv) presents an example of adapting visualizations into a completely new learning situation. The concept for the adaptation along with the example contents were initialized and based on the material by the course lecturer and co-author Essi Lahtinen. The author of this thesis did the technical implementation of adapting the concept contents into the visualization example. The data collection, analysis and writing process were done jointly by the authors.

Publication (v) is a systematic survey of Computer-Assisted Assessment tools between years 2005 and 2010. The literature review and the incremental process for defining the criteria for the tools were done jointly by the group of authors. The writing process was done jointly by all, Petri Ihantola being the corresponding author of the publication.

Publication (vi) introduces the idea and empirical results of the usage of an rubric-based assessment tool, ALOHA, which was developed by a project group part which the author of the thesis was. The publication is mainly written by the author of this thesis while the data analysis was provided by co-author Essi Lahtinen.

Publication (vii) presents a quantitative data analysis of the usage of a rubric-based assessment tool. The data gathering implementation to the tool, data analysis and writing process were done jointly by the authors while the author of this thesis was the corresponding author of the publication.

1. INTRODUCTION

This work is about teaching technologies: *Tools* that help Computer Science teachers and students with their everyday problems when trying to teach and learn programming. In general, tools of a teacher could be used to refer to something more abstract, like assessment approaches, a set of group-work methods, presentation tricks, etc. but in this context, we focus on *tools* that are computer programs that are used by the teacher or by the students to assist in the learning process of becoming a good programmer.

Why are teaching technologies relevant for learning programming? *Firstly*, learning programming is a difficult task where all aid is appreciated. *Secondly*, using computer programs to teach people to write computer programs *just seems so right and a natural thing to do.* Let us take a closer look to the first reason:

Learning programming is a difficult task.

Programming has been stated to be a modern craftsmanship skill. After all, in one level it is a very practical skill with best practices applied to solve practical tasks with a clear output: a functioning computer program. But in order to learn this skill, one needs more than just to practice the skill.

The famous *McCracken study*, or more precisely, *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students* by McCracken et al. [67] from 2001 conducted a large evaluation of novice programmers skills. The results of the study showed that novice students were performing much worse than the teachers expected. The study was re-visited and re-implemented by a similar working group in 2013 by Utting et al. [94] now resulting into slightly more positive results: The students were able to complete the tasks better but most importantly the teachers had more realistic expectations towards their students' (somewhat bad or mediocre) performance.

Learning programming requires a good set of theory on the background—more specifically, not even the kind of theory that is slowly gathered through life all the way from elementary school—but a very specific theory without much relevance to other everyday tasks. General programming concepts are not really naturally

used outside their marginal context, in everyday life, and novice programmers really struggle in forming correct *mental models* of these concepts [82].

Crucial part of adopting programming concepts is the ability to **abstract** to form schemas, or new chunks of knowledge out of them [82]. The creation of computer programs requires working in multiple levels of abstraction: fluently drilling down to technical implementation details of an individual programming language as well as *zooming out* in order to understand the big picture, principles and design of a whole program where the detailed concepts are just part of the implementation details.

Visualization systems can be used to assist in learning programming concepts through abstraction. These systems try to provide additional, visual cognitive input for the user about a foreign topic to help construct a correct mental model. They can help in creating an abstraction of what the lower level implementation details together form and how they together form something new. For instance, when learning *loop structures* the student can use the visualization system to understand that certain syntactical lower level features of the programming language together create a new more abstract concept, the loop structure, which itself is then later used as a tool to solve other problems.

Applying programming concepts into real-life problems is even harder.

Besides adopting a theoretical background one also needs to know how to apply those skills into practice, to learn *good programming strategies and practices*. This is theory on how to apply the practical skills into real problems. In his psychological overview of learning programming, Winslow [98] states that one of the biggest differences between a novice and an expert programmer is that the novice is not able to apply their existing theoretical programming knowledge into real world problems. The same result was also emphasized by Lahtinen et al. [56] in their large international study of novice programmer difficulties. For learning how to apply the theoretical programming skills into real life problems, it is essential to *practice*, *practice* and *practice even more*.

Student programming assignments have a vital task in programming courses. It is the only way for the students to really understand the theory in deeper level, to gain programming strategies and to gain practice on real problems. For the teacher, it is the only way to measure whether the student has achieved a step on the way to the ultimate goal: to become a good programmer.

Assessing programming assignments is more than just checking whether the computer program works correctly. Besides, this itself is **not** a trivial task either[1]. The teacher certainly wants to check this part as well but is also required to look inside

[1] For instance, consider all the malfunctioning/erronous/crashy software in consumer markets!

the functioning program and the process for creating the program: *How has the student reached the goal? Has he/she used a correct programming strategy?* For this task, someone needs go hands dirty and analyse (read) hundreds of lines of program code per student per assignment. In addition, programming courses are often organized as mass courses with hundreds of students with high drop-out rates. The teacher also needs to somehow keep the students motivated enough to keep practicing these skills [2].

Tools-assisted assessment applies computer programs to assist in different parts of the assessment process. **Computer-assisted assessment** tools can be used to automatically test the functionality and other features of a program and rubric-based assessment tools can help the human grader in his task for grading the hundreds of student programs manually and writing constructive feedback for the student.

In this thesis, we will examine the usage and effectiveness of these different tools from the perspective of the whole cycle of a programming courses: Adapting visualization tools so that they support the learning process during the course and best practices in applying tools-assisted assessment.

1.1 Thesis within Computer Science Education Research

From research point of view, this thesis belongs to the discipline *Computer Science Education Research (CSER)*. Over the few decades when computer science education has been actively researched it has grown to be an acknowledged research discipline of its own. In their book, *"Computer Science Education Research"* [28, Chapter 4], Sally Fincher and Marian Petre define:

> CS education research is inevitably interdisciplinary. The nature of CS ... is rooted in mathematically-derived, computational, analytic science. However, the circumstances of the classroom, the nature of education, and models of teaching and learning, are areas that are amenable to investigation only through the human sciences.

In a way, CSER as a discipline is somewhere in the middle of pure Computer Science and pure pedagogical research, and the different research fields within CSER have different positions in the line between these two ends. This *"line"* is rather long with a lot of diversity between the actual research topics. Fincher and Petre [28] divide CSER into 10 research fields:

1. student understanding
2. animation, visualization and simulation

3. teaching methods
4. assessment
5. educational technology
6. transferring professional practice into the classroom
7. incorporating new developments and new technologies
8. transferring from campus-based teaching to distance education
9. recruitment and retention
10. construction of the discipline.

Out from this categorization, this thesis will directly focus on categories 2 and 4. Category 5, *educational technology*, is also very related, as the solutions of this work are technology that improve educational experience. However, in their categorization Fincher and Petre define this category to be also related to presentation systems and smart classrooms, and that sort of more physical education environments where as we will concentrate on software applications. Term-wise this work is exactly about *"technology"*, and more precisely of *"technology that assists in education"*.

The close relation to usage of technology in education is one of those major characteristics that distinguishes CSER from other educational fields. We, *the CS educators*, know technology very well and we are supposed to teach that same technology. So, it is very natural for us to *use the same technology that we teach when doing that.* Pears et al. [77] reviewed the literature around CSER for introductory programming and state that *tools* were the largest category of publications written in the field, as comparison to categories *curricula*, *pedagogy* and *programming languages.*

To yet continue with the placement of the research within the field, Pears et al. [76] in their 2005 ITiCSE Working Group report *"Constructing Core Literature for Computing Education Research"* divide the field into four areas:

A) studies in teaching, learning and assessment

B) institutions and educational Settings

C) problems and solutions

D) Computing Education Research as a discipline.

In this categorization the work falls into Areas **A)** and **C)**: This thesis is contributing technological solutions (teaching technologies) into real-life problems CS educators are facing but we are also reporting on practical experiences and evaluations on those teaching technologies.

The different teaching technologies, or tools, used in CS education can also be divided into subcategories of their own, as presented by Pears et al. [77]:

- visualization tools
- automated assessment tools
- programming support tools
- microworlds.

Out of these, the work focuses on the first two. These are explained in detail in Chapters 2, 3 and 4. Out from the other two tool categories, *"Programming support tools"* refer to IDEs (Integrated Development Environments) that have specific programming support features like interactive incremental code execution, visualizations, and editing and syntax support. A well known example of a such an IDE-based tool is **BlueJ** [47]. Similar ideas are overlapped in the field of visualization tools, for instance the interpreter-based **VIP** tool [97] has also a simple code editor, but definitely falls more into the category visualization tools category instead of an IDE with visualizations. *Microworlds*, then, are their own environments where the student is supposed to control part of the environment by programming. One example is **Karel the Robot** [75] in which the student is controlling a robot in a world of streets and intersections.

1.2 Research Questions

The one main research question what this thesis is contributing is:

How can the teaching of programming be effectively assisted using teaching technologies?

Here, that main question is divided into smaller, more concrete ones that are directly addressed in the included publications. Out of the different kinds of tools this work focuses on two major categories: visualization systems and tools-assisted assessment.

The research on visualization systems is a wide area with a lot of different focus areas. The different areas are introduced in Section 2.4. In this thesis, the focus for visualizations will be around the following research question:

- **RQ 1: How can visualization systems be well integrated into programming courses?**

  As we will discuss in the Chapters 2 and 3, proper integration of visualization system into the course is vital for the success of enhancing learning and facilitating teacher workload. Proper integration itself includes both consistent usage during the course over time and learning materials but also having visualizations that engage different cognitive levels.

  Thus, we will divide **RQ 1** into the following concrete questions that are addressed by the publications of this thesis.

  - **RQ 1.1** How can program visualization examples support also deeper learning?
  - **RQ 1.2** How to integrate visualization tools into student homework assignments?
  - **RQ 1.3** With proper integration from both cognitive and practical perspective, are visualization systems helping to learn programming?

The second larger research question is then focused around the assessment process of programming courses. As we will discuss in Chapter 4, assessment plays a major role in guiding the student activities and focus. As teacher workload is the practical limiting factor of *ideal assessment* we will seek aid from different kinds of tools. Our research question for assessment is:

- **RQ 2: How can the assessment process be assisted with teaching technologies?**

  We will look into a process called semi-automatic assessment that integrates both computer-assisted assessment and manual assessment. Both of these can be facilitated with tools and this thesis will study the following the following concrete questions:

  - **RQ 2.1** What kind of teaching technologies there are in terms of features for the computer-assisted assessment of programming assignments?
  - **RQ 2.2** How can manual assessment be assisted with rubric-based assessment tools?
  - **RQ 2.3** Are rubric-based assessment tools effective and being used as they ideally should be?

To answer these questions this work presents surveys of different teaching technologies and related literature that are addressing the problems in the field as well as empirical quantitative data analyses on their effectiveness. We will present adaptations and extensions of existing learning taxonomies into practical use and to validate our results, we have collected empirical data and present statistical analyses on their usage. The individual research methods used for these tests are described in the corresponding publications that are part of this thesis.

1.3 Organization of the Thesis

This thesis is a combination of seven peer-reviewed academic publications and an introduction part explaining the problems that are addressed, summarizing the surrounding research field, the contribution of the included publications' to the research question and relevant conclusions.

After this introductory chapter, an overview of visualization systems is given in Chapter 2. Chapter 3 discusses what is deeper learning in terms of CS education and how visualizations can support that. Chapter 4 describes the assessment process of programming assignment and the use of tools in facilitating it.

After these introductions, Chapter 5 summarizes each of the included publications of this thesis. Chapter 6 gathers the results of the thesis from the publications into one place addressing the research questions, summarizes the conclusions and discusses the limitations, generalizability and benefits of the work.

At the end of thesis, the original included publications are reprinted.

2. INTRODUCTION TO VISUALIZATION SYSTEMS

In this chapter, we will give an overview of the existing usage of software visualization in programming education and their effective usage in teaching introductory programming. We will start with looking into what visualizations are, continue with algorithm visualization systems and then move on to program visualization systems and their effective integration into programming courses.

2.1 What Are Visualizations?

As *visualization* is a term that is naturally associated with something *visual* and related to drawing pictures, it is necessary here to clarify the term from the perspective of Computer Science Education Research. While containing visual elements, the visual appearance is not necessarily the purposeful output of a visualization but the focus is more on creating a certain *mental model*, or a *mental picture* through multiple forms of sensory input [93]. In all, with visualizations, we are talking about a pedagogically enhancing combination of visual, textual and possible even audio cues that help student create a correct mental model of the behaviour of computer program or certain part or aspect of it.

The general term that is used to refer to all kinds of visualization aids describing a computer program is **Software Visualization (SV)**. A software visualization is something which can assist all kinds of software developers in writing, analysing, testing, debugging and optimizing code but can also be used as a pedagogical aid in the CSER context [28]. To give common terminology for software visualization, Price et al. [79, 93] define a widely-adopted *Taxonomy of Software Visualization* that divides SV into two subcategories, and their respective subcategories:

- **Algorithm Visualization (AV)**, further split into Static Algorithm Visualization and **Algorithm Animation (AA)**. The first ones refer to any static visualizations of a given algorithm, for instance flowcharts, where as AA refers to dynamic visualization of an algorithm by means of a video or, more relevant to this context, an algorithm animation system, a specific software. An example of an AV system is the MatrixPro, shown in Figure 2.1 a bit later in this chapter.

- **Program Visualization (PV).** While the purpose of AV is to visualize a principle and behaviour of an algorithm in an implementation-independent way, animating the flow of the data structures and the idea of the algorithm in a higher level, PV is used to visualize *the implementation details* of a given programming task, such as an algorithm written in **one** language [53]. A PV tool would animate the individual expressions, function calls and the state of memory allocations of the implementation. PV can be used in the teaching of very introductory programming to demonstrate even the simplest programming concepts such as conditional or loop structures. An visual example of a PV tool would be Jeliot 3, shown in Figure 2.3 a bit later in this chapter.

The creation of a visualization tool usually originates from the need of an individual instructor and the educational needs focus either on algorithms or basic programming. As the abstraction level of teaching those subjects is different, it can easily guide the creation of the tool to one way or the other. The exclusive division between AV and PV tools has become slightly artificial and blurred as many tools are capable of doing the both [50]. The difference, based on which the division is done, comes from one or more of the following features:

- Is the visualization focusing on the data structure and its changes during the algorithm (AV) or showing lower-level structures such as memory contents and function call stacks (PV).

- Is the code that is visualized pseudo-code of an algorithm (AV) or an individual programming language (PV). Certainly AV tools can also show the implementation in an individual language and PV tools can operate in multiple languages.

- Is the tool used in the introductory programming course (PV) or on an algorithms course (AV)

- Is the research around the tool focusing on AV or PV, or more simpler, *How do the authors define the tool themselves?*

Stasko et al. [93] define the following roles for people creating and using software visualizations: A *programmer* is the person who has written the piece of code, or algorithm, that will be visualized. Programmers may or may not know that their code will be visualized. A *SV software developer* has written the software that allows programs or algorithms to be visualized. A separate role from this is the *visualizer* or *animator* that takes the code, the SV system and specifies how the visualization is to be connected or applied to the code. Finally, the *user* or *viewer* will view the visualization or interact and navigate through it.

Each of these roles have different interests and expectations from an algorithm animation system. *Users*, or *viewers* of the animation are interested in the user interface and user interaction of the tool. *Visualizers* require features that facilitate effortless and flexible animation creation to match personal preferences or experience levels. *SV software developers* may then be interested in responding to those requirements coming from visualizers or users by updating or extending the tool easily. [84]

The above-mentioned roles can be overlapping and the same person can act in multiple roles. The typical use case with programming learning situations is that the student acts as the user of the visualization and the course instructor is left with the rest of the roles [43]. Especially in the context of algorithm animations it is possible that student engagement is enhanced by having the student also take the role of visualizer, creating visualizations for other students to view. This will be discussed more in the Chapter 3 where we will look into the Engagement Taxonomy and ways of enhancing deeper cognitive levels through visualizations.

2.2 Algorithm Visualizations

Let us begin with an overview of tools around Algorithm Visualizations, especially focusing on Algorithm Animations with a tool, as that is where the research in the area historically originates from.

Most of the historical overviews on research or theses in this area start by mentioning **"Sorting Out Sorting"** [11], a 30-minute teaching film from 1981 about nine sorting algorithms. This thesis is not an exception because of the following.

"Sorting Out Sorting" is worth starting with because, *firstly*, it clearly addresses the problems teachers have when teaching algorithms or program behaviour using traditional, more analogue tools like blackboard, or nowadays, whiteboards and slide presentations: As programs are temporal, executing over time, their *behaviour* can be better represented using *animations* of their state and state changes than with static images. Especially algorithms[1] include a lot of repetition.

Secondly, "Sorting Out Sorting" also shows that with well-designed visual cues a successful visualization can also be a video, *or rather a movie in this case*, not necessarily a separate software tool. However, we will get back into the importance of student engagement later in Chapter 3. And *finally*, besides being a fun piece of work, the video is still being used and at a time, over 30 years ago, created a good basis and inspired instructors to get interested in ways of visualizing program behaviour.

[1] Algorithms do not refer **only** to *sorting algorithms* that often are taught in more advanced CS courses but usually any problem solved with a computer program involves an algorithm of some sort.

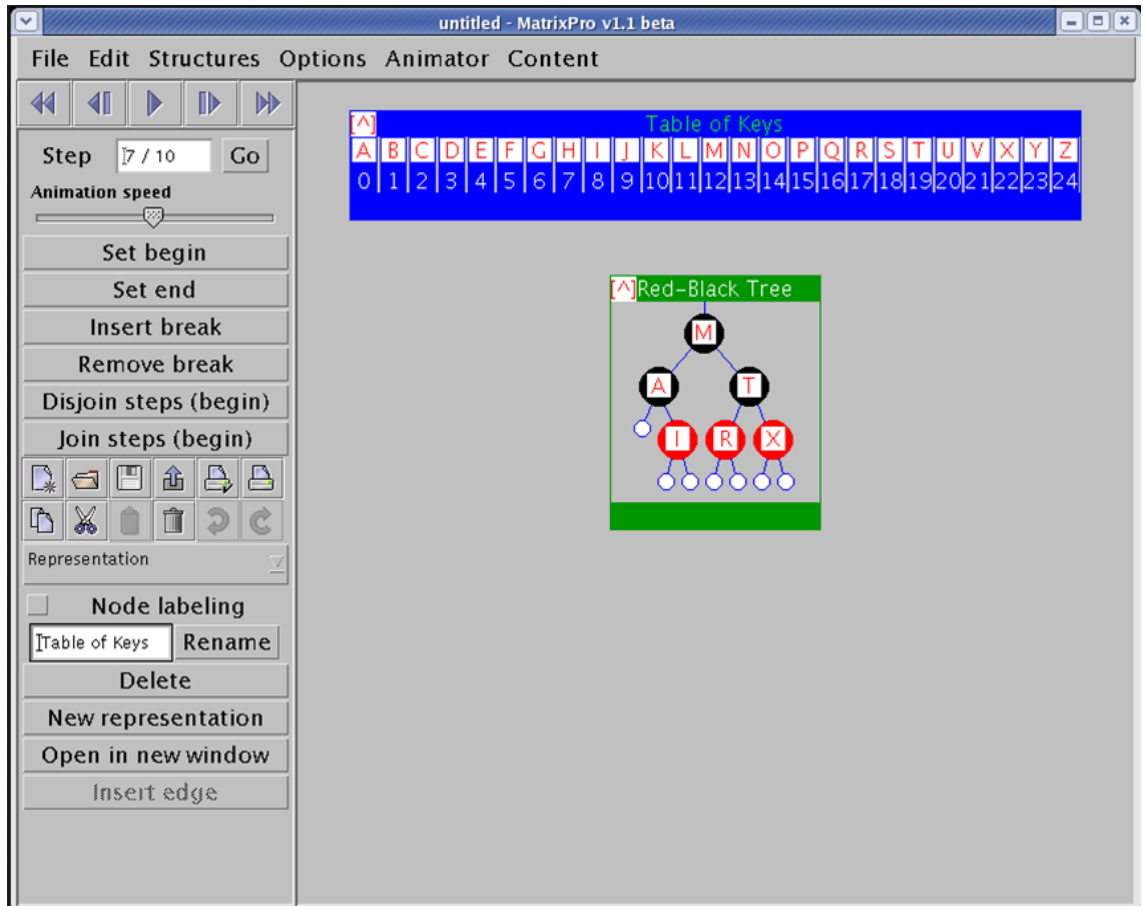


Figure 2.1: The MatrixPro Algorithm Visualization tool, demonstrating the behaviour of a Red-Black Tree.

The early research around algorithm visualizations often mentions at least the visualization systems **BALSA** [20], which provided multiple dynamic views into the algorithm and highlighted *interesting events* through a separate animator, **Zeus** [19], that used also colors and sound, and **Tango** [92] with its X Windows-based follow-up **XTango** [90], that introduced *the path-transition paradigm* for *smooth animation*.

As for the more recent tools that often appear in relevant literature and are based on previous research on the field, one good example is **ANIMAL** [84], that was especially designed to provide relevant features to support multiple of the visualization roles, defined by Stasko et al.[93]. In practice, ANIMAL was designed to have a good, interactive and customisable user interface for the users, visualizers had multiple approaches for creating graphical representations (GUI, scripting using ANIMALSCRIPT [83] or an API) and developers were able to extend the system, for instance with new programming languages, without touching the core system code.

There has also been a lot of research around the **TRAKLA2**[2] framework [66] which has been widely adapted into use by various Finnish CS faculties. TRAKLA2 is using the **Matrix** [49] algorithm animation and simulation engine, later succeeded by **MatrixPro**[3] [45] for the graphical representation. An algorithm visualization example of a Red-Black Tree within MatrixPro is shown in Figure 2.1. The TRAKLA2 framework itself lets teachers create interactive exercises from generated values that can be automatically assessed by the system for their algorithms curricula. The learner can also examine the model solution of an exercise. Using the system students can learn and practice algorithms as well as complete and submit exercises as part of their coursework. So instead of just being an AV tool, TRAKLA2 is a coursework framework (utilizing the MatrixPro AV tool) that can be easily adopted and integrated into the whole algorithms course as an effective study mean.

Over the years, a plethora of different algorithm visualizations and animations have been created, but they lack centralized distribution and coordination. In general, the quality of them has been unfortunately often poor and mostly concentrating on the easier topics [86, 87]. A good place to look for an overview of the up-to-date status of algorithm visualizations and search for individual examples, tools or research is the **AlgoViz portal**[4], that has been centralizing AV related resources under one portal for quite a few years.

2.3 Program Visualizations

As Algorithm Visualizations concentrate on visualizing general algorithm behaviour in a higher-abstraction level without going into implementation details in a specific language, Program Visualizations connotate connections within a program in a lower level, concentrating e.g. on the variables, data structures and function call stacks of the program. Program Visualizations itself contain different kind of approaches, defined by Price et al [79] and adapted by Sorva [89]:

- Static program visualizations that represent program code with dependencies and code evolution.
- Dynamic program visualizations, illustrating program runtime behaviour, containing *Program animation*, where program visualizes what happens during a program execution, like a *Visual Debugger*.
- Visual Programming, where programs are specified using graphics rather than visualizing a program written in a non-visual language. As a separation to

[2]http://www.cse.hut.fi/en/research/SVG/TRAKLA2/
[3]http://www.cse.hut.fi/en/research/SVG/MatrixPro/
[4]http://www.algoviz.org

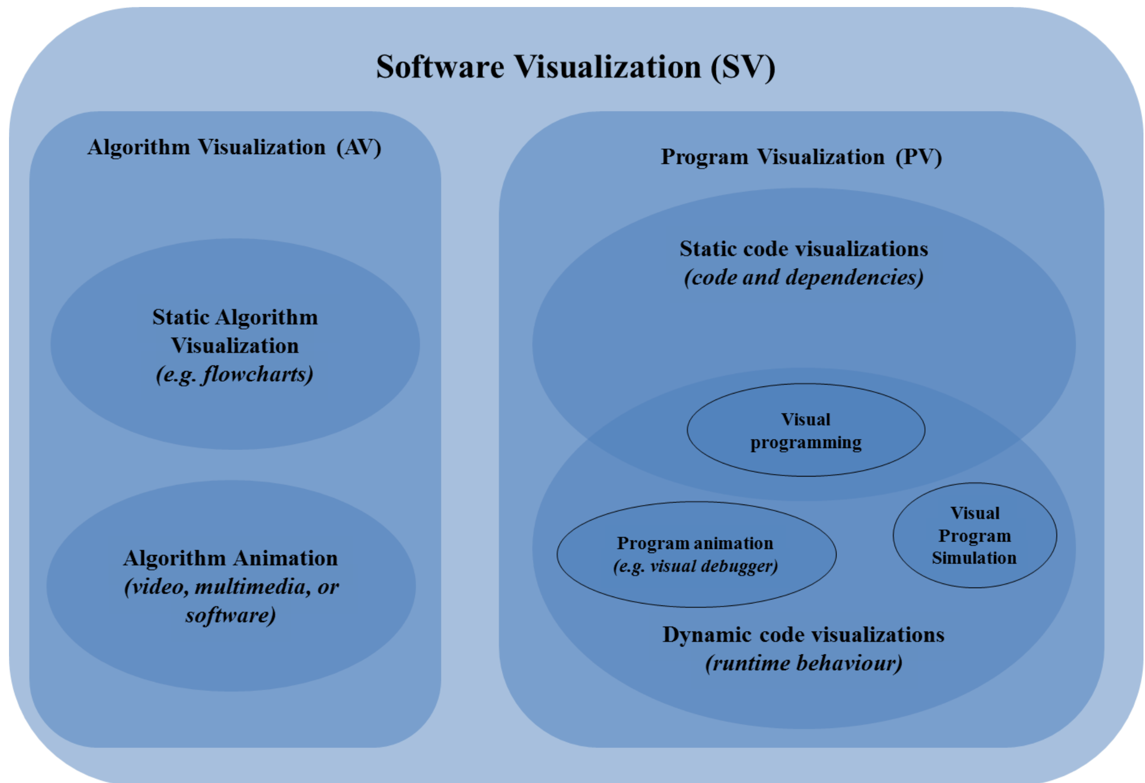


Figure 2.2: Different types of Software Visualizations, based on the version of Sorva [89] which is adapted from the original categorization of Stasko et al [93]. Proportions are not here relevant and intersections have been simplified.

other parts of SV, the purpose is to define new programs instead of making existing programs easier to understand.

- Visual Program Simulation (VPS). A pedagogical approach developed by Sorva [89] for immersing students in the dynamics of program execution by visualizing a notional machine, and engaging the student during an activity. The main difference to a visual debugger is that where a visual debugger finds automatically what is happening and illustrates that to the user, VPS leaves the user to define what is going on in the program execution.

The whole wide field of visualizations systems is illustrated in Figure 2.2. The PV systems used in education are mainly dynamic program visualizations and type of visual debuggers that have been improved by the teachers to be used in pedagogy [89]. The main purpose of the PV tool is to provide additional visual assistance to the learner to understand the program code better.

One of the most known and widely-researched PV tool is **Jeliot**[5]. The stages, evolution and research around Jeliot are described in an article by Ben-Ari et al [15]. Jeliot-family is based on the very early **Eliot** system [58] that was intended

[5] http://www.cs.joensuu.fi/jeliot

to visualize algorithms written with C programs. Jeliot is specifically designed for learning elementary programming pioneering in automatic animation so that the student or teacher does not explicitly need to build the visualization of a given source code. This is specifically important for PV where the purpose is to visualize large number of smaller programs where as algorithm animations typically focus only on a limited number of existing algorithms. The creation of algorithm visualization examples can take more time, and be based on separate scripting languages as the examples are reused multiple times.

The latest evolution of the Jeliot-family is **Jeliot 3** [69]. It is based on a full Java interpreter which makes the tool almost fully compliant with full Java language. The main user interface of Jeliot 3 is shown in Figure 2.3. On the left side of the screen, the animated code is being highlighted as executed with the VCR-like controls in the bottom. The right side *Theater* automatically animates the data structures, evaluated expressions and executed methods visualizing the call tree of the functions. One of the main principles of the Jeliot animations are around consistent and complete animations: each evaluation of expression and subexpression is displayed.

VIP[6], or Visual InterPreter, is a program visualization tool developed and researched in the Edge-group[7] of Tampere University of Technology (TUT). The original implementation was developed by Antti Virtanen as part as his Master's Thesis [96] in 2004 and then further improved in two parts, first re-creating the underlying interpreter engine (**CLIP**) [64, 65] and then re-creating the actual visualization tool [40] in 2009.

VIP was designed to facilitate the learning of introductory programming in basic imperative C++ in Tampere University of Technology. Instead of showing static, more or less hard-coded visualizations of a C++ example, it was based on a real programming language interpreter, thus making it visualize whatever was written in the program code, same way Jeliot 3 does with Java. This enabled the instructor to easily create material by just providing simple imperative C++ examples to the tool [97]. Making an interpreter for the full (or even close to full) C++ standard supporting for instance object-oriented programming was not feasible and especially not needed in the very first programming courses. That is why VIP works with a subset of C++, namely C–. The language allows the basic usage of imperative C++ within the tool, for instance data types, loop-structures, and functions.

VIP provides multiple simultaneous views to a program code. The main user interface of VIP is shown in Figure 2.4. Besides the code window, the user interface presents a panel for guiding the execution of the program, a variable view with

[6]http://www.cs.tut.fi/~vip

[7]Edge is a Development Group for programming Education, http://www.cs.tut.fi/~edge

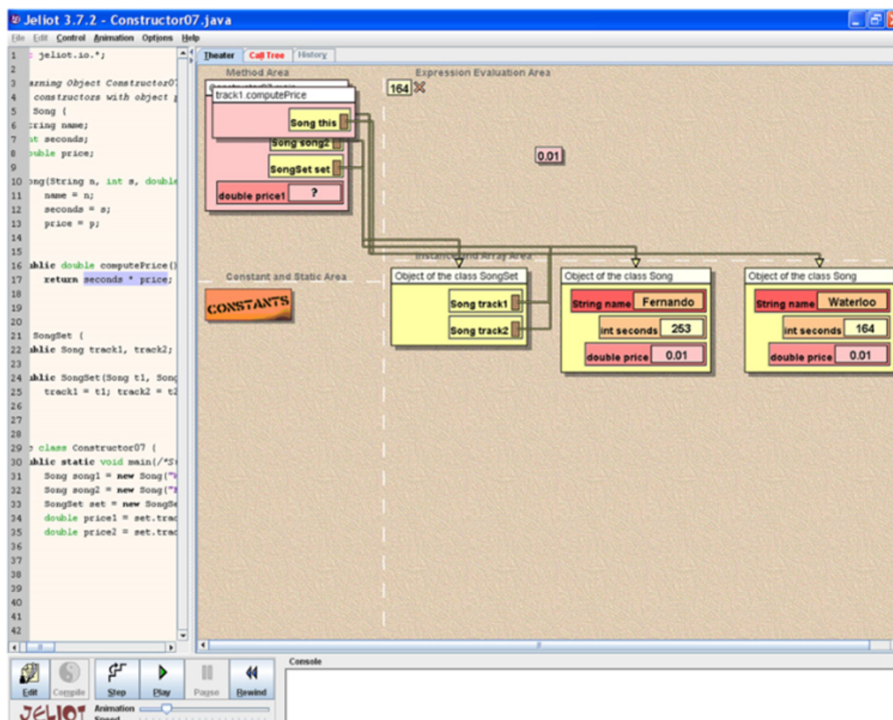


Figure 2.3: The main user interface of Jeliot 3, taken from [15].

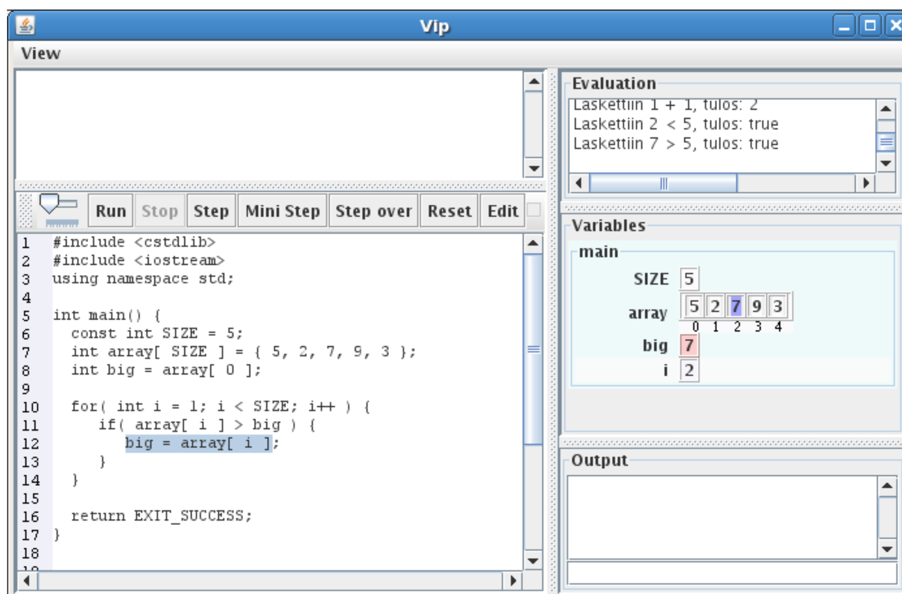


Figure 2.4: The main user interface of VIP.

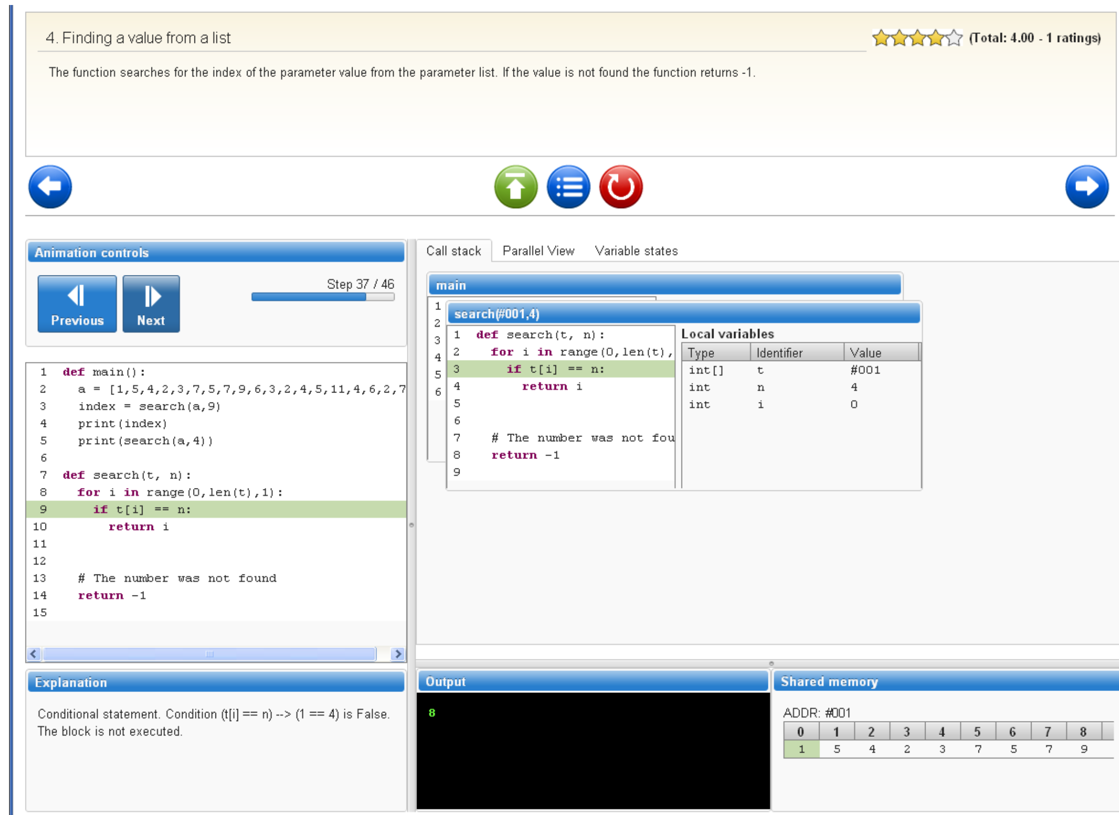


Figure 2.5: The visualization view of ViLLE PV tool.

current values and the function call stack, output view, a separate evaluation view and a view for additional textual explanation of the current execution step. The code itself is given for the interpreter as a normal C++ source file but with specially annotated additional comment lines to provide the textual cues for the execution. The student can also jump into the a separate code editing window to modify the visualized code himself.

A widely-adopted PV tool and learning environment of recent years is **ViLLE**[8] [81], developed at the University of Turku. Originally the purpose of ViLLE was to become a PV tool to facilitate the everyday work of teacher by providing a way to visualize elementary programming concepts without direct dependency to an individual programming language so that the student can see the same example in different programming languages. The main visualization view of ViLLE is shown in Figure 2.5. Since past ten years, the tool has grown to a full learning environment that can automatically assess student exercises [41], promotes collaborative learning [80], provides multifaceted ways for collecting research data, and supports virtual badges and gamification features. Besides adult education ViLLE has been adapted successfully to high school and lower-level education.

[8] https://ville.cs.utu.fi/

One of the latest evolutions of PV tools is **UUhistle**[9] (pronounced *'whistle'*) that is a PV system specifically designed to enable also usage as Visual Program Simulation (VPS). The tool and VPS are thoroughly described in the thesis written by Sorva [89]. UUhistle visualizes a *notional machine* for the Python programming language, supporting a suitable subset of the whole language. It visualizes program code execution, expression evaluation, the call stack and the state of memory. UUhistle can be used as a regular PV tool for learning Python through its collection of predefined examples or any custom content. The tool can also ask interruptive questions about the execution of the program engaging the student and the teacher can create full exercises using these features.

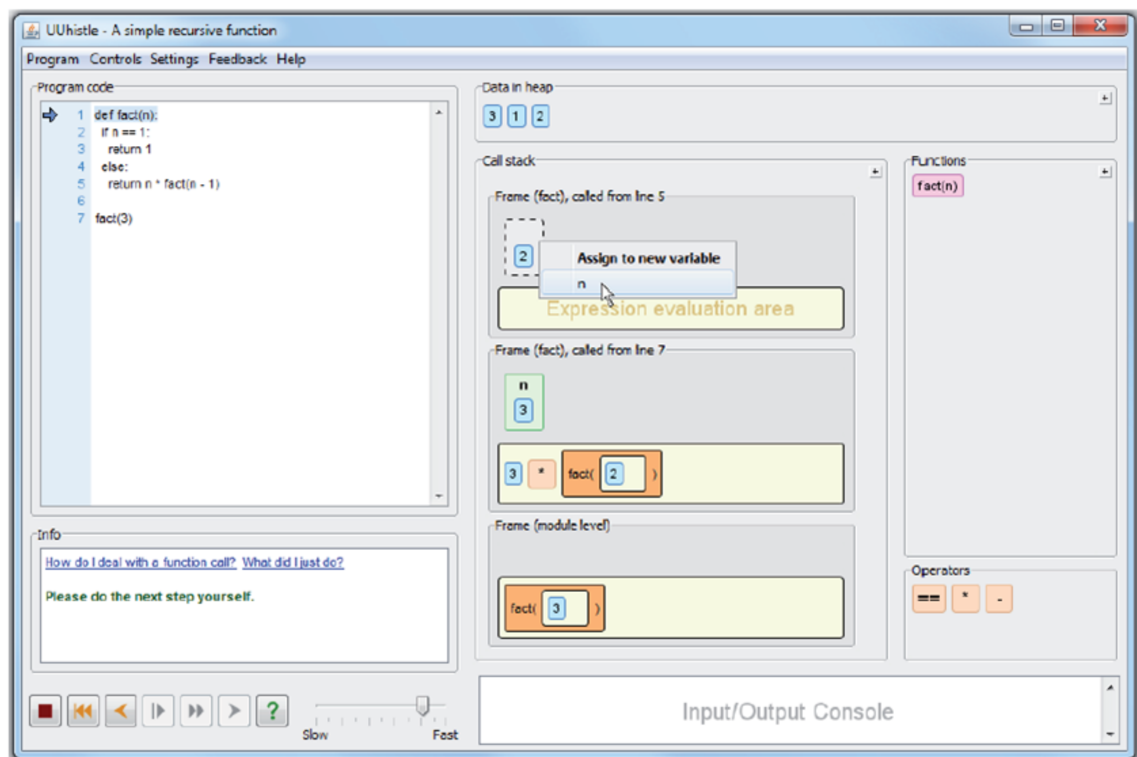


Figure 2.6: The UUhistle PV tool, here running a Visual Program Simulation example. The user is supposed to simulate the given program code himself using the interactive visualization GUI. Here, he has just dragged a parameter value into the topmost frame and is just about to name the new variable using the context menu. This example and screenshot are taken from [89].

What makes UUhistle advanced compared to other PV tools is the ability to turn examples into Visual Program Simulation exercises. In a VPS exercise the user is engaged so that the user is given the program code for which he is supposed to simulate the execution himself by interacting with the visualization components. An example screenshot of a VPS exercise in UUhistle is shown in Figure 2.6.

[9]http://www.UUhistle.org

2.4 Usage and Effectiveness of Visualizations in Education

The effectiveness of visualization systems has been widely studied in different use cases over the years with very varying results. The main focus areas of relevant research over the time can be divided roughly into:

- early studies on AV effectiveness
- integration to course infrastructure
- effects in cognitive process of the student
- how visualizations are used
- teacher's attitudes towards visualizations.

We will now look into the most notable findings in existing research of these areas to create a validated basis for our further assumptions.

2.4.1 Early Studies on AV Effectiveness

In the light of earlier, slightly disappointing and mixed studies about Algorithm Visualization effectiveness (e.g. one of the very first studies by Stasko et al. [91]), Kehoe et al. [46] conducted a study where they evaluated the usage of AV utilized more in a "homework" learning scenario rather than isolated "final exam" scenario. They concluded that the pedagogical value was better in open homework sessions as students need human explanation to accompany the animations. They also discovered that the use of AV enhanced the motivation making the algorithms less intimidating and thus enhancing student interaction with the materials and facilitating learning.

A good overview of earlier research on the topic is presented by Hundhausen et al. [33] through their systematic meta-study of 24 experimental studies of the effectiveness of algorithm visualization systems. Their most significant finding is that an greater impact on effectiveness is based on *how* students use the tool instead of *what* is shown them by the AV tool.

2.4.2 Integration to Course Infrastructure

Ben-Bassat Levy et al.[60] created the **Jeliot 2000** (later succeeded by **Jeliot3** tool [69]) and evaluated its use in a programming course for 10th grade students with a teaching setup where two parallel 10th-grade student groups were taught basic programming with same materials except that one class was also using the PV tool. In their study, consisting of pre- and post-tests as well as interviews they found out

that the already-well-performing students do not need visualizations, however they were not *harmed* by its use either, the poorly performing ones can be overwhelmed by the tool, but majority of the students in between gain a lot from the usage of tool. Besides scoring better, the group that used the PV tool also showed problem-solving techniques adopted from the tool as well as the right terminology that is important from socio-linguistic approach to learning. An important conclusion of the study was that "[the PV tool] *must be integrated into the classroom and assignments, rather than used as a one-time teaching aid."*

Proper integration to course materials and to the whole flow of the course is a key factor in the success of adopting a program visualization tool into a course. Instead of just covering all the topics they should cover all the different learning situations and cognitive levels related to the subject [51]. In one of our own studies [55] around the usage of PV in a programming course we achieved positive results on students' voluntarily use of the tool when the PV tool was used throughout the whole course setting in a consistent way .

Similar kind of results were also found by Kaila et al [42] where they examined the use of their **ViLLE** PV tool during three consecutive years of the same high school programming course. During the first two years, students were introduced to ViLLE only in the beginning of the course. On the third year, ViLLE was used consistently throughout the whole course. The students on the third year performed statistically significantly better than in the first two years leaving the researchers conclude that the success was, at least partially, due proper, consistent integration of the PV tool to the course set-up.

2.4.3 Effects in Cognitive Process

Ebel and Ben-Ari [23] state that quite many studies in the effectiveness of visualization tools in pedagogical aspect have *"a weak point in that they represent what the student* ***says****, which is not necessarily indicative of his or her real mental state"*. As in order to bypass the student-link in the studies they conducted a research on the students' attention because it correlates with learning effectiveness. This was done by unnoticeable video taping of lessons where PV was used and observing the student behaviour in class. More precisely, a certain kind of behaviour was measured, the amount of ERUA (Episodes of Recognizable Unattentive Attide), such as teasing the teacher or other students, throwing erasers etc. When the amount of ERUA was the smallest the students were most attentive and thus learning the most.

To magnify the results results of the study of Ebel and Ben-Ari [23], the group of students was part of special education because of emotional difficulties and learning disabilities like ADHD. With this kind of group the amount of ERUA is typically

not small, but, in this case, during classes when PV was used, the amount of ERUA did not only lessen but ceased completely. For generalization of the results outside special education the authors state that *"The best students do not need it [PV] and the worst are not helped. — [S]ince improved attention and behaviour can be obtained [within special population] through the use of PV, even in a normal class students .. are likely to be helped."*

As PV tools often have multiple simultaneous presentations in adjacent views, eye-tracking is one way to empirically study what the learner is actually focusing when interacting with the tool. Bednarik has conducted a series of studies around effectiveness of visualizations using eye-tracking mechanisms, some of them concluded in his doctoral dissertation [13]. In one of these studies, Bednarik et al. [14] conducted empirical eye-tracking experiments on students using the Jeliot 3 system. They investigated how experts use the tool in comparison to novice students. Their results show that experts read the code first and used the animations only to test their hypotheses, while novice students relied on the visualizations without reading the code first, interacting much more with the tool. Analysing the gaze patterns of the students they also conclude that visualization tools could be designed to be more *adaptive* in order to reduce the cognitive load of the user as the user becomes more experienced with the topic.

Nevalainen and Sajaniemi [74] present a model of the cognitive phenomena that takes place when using a PV tool. Using multiple experiments, for instance eye-tracking, they studied where students paid attention on the tool and their short-term and long-term mental models on a given subject, in this case the different roles of variables in an application. Their results showed that students were spending surprisingly small amount of time watching the visuals or animations and relying mostly on the textual cues present on the screen. As a conclusion they emphasize proper design of the PV tool user interface: in absence of step-by-step animation of the program behaviour the students focused even more on program code and textual cues instead of the static visualizations.

In her studies around visual programming and the use of graphical notations, Marian Petre [78] discusses the importance of secondary notation: the use of layout and perceptual cues to guide the structure of the visual representation. She found out that expert programmers had generally better and more coherent strategies in navigating through problems, mainly using the text to guide how they approached the graphics and being able to grasp on those secondary notations in order to structure and approach the graphics. The same way, the experts would create visualizations in more organized and easily-approachable way. However, she also points out that even though secondary notation is important, the less skilled programmers, that often benefit of visualizations, would not be able to fully benefit of it because of

poorer readership skills. Reflecting these results into the beneficial use of program visualizations we can state that automated visualization systems can sometimes create possible issues by providing 'bad' secondary notation because of automation, for instance, laying out visual items in an un-organized way, mis-cueing and possibly causing confusion. So especially with higher-level visualizations examples, the teacher needs to pay attention into having examples that are not only functioning correctly but also 'look' logically correct. As an expert programmer, the teacher grasps these relationships rather naturally.

2.4.4 How Visualizations Are Used

Lahtinen et al. [57] conducted a large international survey for 335 novice students and their teachers who had used visualizations as extra material in their programming courses. The purpose of the study was to examine the different student groups' attitudes towards PV and the use cases where voluntary use of PV systems were most helpful for the students. The students had mostly used visualizations while self-studying but in general found them most useful when the teacher presented them. The study identified two use cases where PV was perceived most useful: Moderately successful students were able to benefit of PV during their independent work where as less successful students found PV useful when the instructor guided the use for instance during a lecture or exercise session. Based on the results the authors remind teachers to focus on the way visualizations are presented and how their usage is guided for the students.

Isohanni and Knobelsdorf [35] conducted a thorough qualitative, *student-oriented*, research on *how* TUT students use VIP. Instead of trying to measure the learning outcome they focused on identifying the working patterns the students followed when solving problems with the tool, and how that correlates with what was taught to them about the tool usage. Besides dynamically interacting and debugging the issue with the tool students also used the tool only to encounter an issue in the code and then performing the actual analysis either statically with whatever was already visible in the screen. Some even abandoned the tool at this point to continue with more traditional, analogue methods.

2.4.5 Teachers' Attitudes towards Visualizations

From a wide survey to CS educators on their use of visualizations by Naps et al [73], they got an overwhelming result with nearly all (97%) respondents being convinced that visualization can make a difference in helping learners better learn concepts. However, the major doubts towards visualizations were related to the time required from the teacher. Also problems with practical, consistent use of visualizations were

observed or as the authors nicely phrase this: *"even though [many computing educators] use visualizations, the visualizations are not really woven into the instructional fabric of the course."*

Ben-Bassat Levy and Ben-Ari [59] conducted a phenomenographical study on teachers' attitutes towards the Jeliot PV tool. Their outcome space *(i.e. the result of a phenomenography)* has four different categories:

- dissonant *(interesting tool but reluctance to use it)*
- rejection *(not found useful)*
- by-the-book *(possibly useful tool, but may not suitable for teacher's case)*
- internalization *(useful tool).*

Ben-Bassat Levy and Ben-Ari [59] state that the negative categories of their outcome space could be the result of two matters: *First*, tool has not been properly integrated into the curriculum through other learning materials. *Second*, centrality: The teachers might feel the tool is threatening their position in the course in some way, e.g. "I'm an experienced teacher and do not need such a tool to help me in teaching".

In a recent study by Isohanni and Järvinen [34] conducted a survey for that was answered by 255 programming teachers from 33 countries to find out how much visualizations are used, how they are being used, and what are possible reasons for not using them. Nearly half of the courses did not use visualizations at all and regular use of visualizations was found to even more rare, only around 20 per cent of programming courses utilize SV regularly. Also, on contrary to the recommendations they are mostly used by only the teacher and not by the students. The main reason for using visualizations in the survey was that they found the tools educationally effective. Other reasons for using mention the easy-of-use and the possibility to create own visualization examples with the tools. The main reason for *not* using visualizations was that teachers preferred using their own visualizations, e.g. drawing pictures on the black board. Also for more advanced courses, the teachers did not necessarily believe in the educational effectiveness of visualizations or that there would have been visualizations available for their topic.

2.5 Summary

To summarize the existing research presented in the above sections, we can conclude the following about visualization systems in programming education:

- Visualization systems in educational usage are usually divided into Algorithm Visualizations and Program Visualizations [79]. Even though focusing on different abstraction levels, they share the same purpose and mainly focus on

same larger issues: enhancing the cognitive process of the student when learning programming and facilitating the course infrastructure.

- There are a lot of existing tools and individual visualizations available, but a common centralized organization is missing and the quality of the existing examples varies a lot [33, 86, 87].

- Looking at the situation over time, these tools have, however, become more advanced, emphasizing better engaging and developers of these tools have taken the findings of previous research into account[10]. Also, a lot of international collaboration between academics exist in the field, trying to make it more coherent. [87]

- There are a lot of studies where visualizations have improved the learning outcome of students, but also studies where no significant difference has been observed [33]. Nevertheless the use of visualizations has at least contributed positively on student motivation and time spent on studying [55]. Visualizations also improve students vocabulary and terminology on the subject, and especially when used collaboratively, they engage students into discussions of better quality [60].

- It is important that the visualization tool is well-designed so that it does not distract the student but provides relevant information, guiding animations, sufficient textual cues and especially engages the student [74, 43]

- Teachers' attitudes towards visualization systems vary from very positive to not even knowing about them. As knowledge and experience from using them increases their attitudes become more positive. [34, 59]

- Visualizations work better when they are used consistently during the course, not only over time but also over various means of teaching and learning [51, 42, 60, 59]

Clearly, one of the key factors for successfully using visualization systems is in the proper integration of the system to the whole learning process. It is important *how* and *when* the students use the tool and how they engage with the tool. The next chapter will discuss the importance of engagement in visualization examples so that they can enhance deeper cognitive development and thus helping in efficient integration to programming courses.

[10]This is of course tautologically implied to only the tools that have own research around themselves. Of course, there can be an army of non-researched tools completely developed in a void of their own, disregarding everything ever written about good SV tool design.

3. TOWARDS DEEPER COGNITIVE LEVELS WITH VISUALIZATIONS

As discovered in the existing research introduced in Chapter 2, successful adaptation of visualizations into a course requires proper integration to the flow of the whole course. Besides integrating to materials and different learning situations it also means integration to different stages of learning. Instead of just introducing new concepts, visualizations can also be used to deepen the understanding of a subject when engaged more with the visualizations. This chapter introduces two different learning taxonomies that give us tools to talk about deeper learning: the Engagement Taxonomy and Bloom's Taxonomy of Cognitive Development.

3.1 Engagement Taxonomy

We will start with an introduction to the Engagement Taxonomy that is a taxonomy specifically describing the level of engagement of a visualization system in relation to its pedagogical effectiveness.

Hundhausen et al. [32] discovered an important relation between learning and student engagement when using a visualization system. In their study, the students were supposed to use an Algorithm Visualization system to construct and present their own visualizations. The assessment of effectiveness was based on Social Constructivism and instead of controlled experiment through learning outcomes *ethnographic field study* consisting of multiple field techniques was conducted. The study showed that the AV software was actually distracting the students from concentrating on the relevant activities as, in this case, their focus shifted away from *learning the algorithm* towards *learning how to program graphics*. However, when the students were creating the algorithms using low-tech art crafts in a "storyboard" way for presenting and teaching the subject to others, the students got more participated into the course and their presentations stimulated more relevant discussions about algorithm concepts. This indicated that student engagement has an important role on the effectiveness of visualizations: Instead of being a *knowledge conveyor* the AV technology became a *conversation mediator* contributing positively on the learning experience.

An ITiCSE Working Group in 2002, led by Tom Naps and Guido Rössling, conducted a large study on the use of visualizations amongst CS educators. In their

Table 3.1: Levels of the Engagement Taxonomy [73].

Form of engagement	**Explanation**
1. No viewing	There is no visualization.
2. Viewing	Basic requisite - there is a visualization to view, at least. Either passively (e.g. just watch a video in a class room) or actively (have controls to the visualization).
3. Responding	Student is required to answer questions concerning the execution of the visualization, e.g. *"What will be happen next in the visualization?"*, *"Is the algorithm free of bugs?"*
4. Changing	Modifying the visualizations. Student can change the input data set of the algorithm.
5. Constructing	Students construct their own visualizations of the algorithms.
6. Presenting	Presenting a visualization to an audience for feedback and discussion. These may or may not be created by the students themselves.

report [73], they present results of the study and propose actions on increasing the effectiveness of visualizations.

By the survey responses and literature studies the working group states that *"such [visualization] technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity."* As a result they suggest a new taxonomy of learner engagement, called the **Engagement Taxonomy (ET)**, presented here in Table 3.1.

In a way the taxonomy is hierarchical, with levels building on top of each others, but not necessarily requiring that. The levels can also be achieved individually, with the exception that *Viewing* is a a requisite for the last four levels.

As the Engagement Taxonomy is originally designed for Algorithm Visualizations it is well adapted and spread by AV tool designers, for instance, [44]. The same underlying principles around ET can and have well been applied into program visualizations as well. Here we present some of the follow-up work around visualizations systems, based on ET.

A system called **JHAVÉ** [72], developed by Tom Naps, is not so much an AV system of its own but an environment to support existing AV systems by improving engagement in their usage as suggested by the Engagement Taxonomy. Existing AV engines can be plugged into it and JHAVÉ provides for instance controls for interacting with the animation engine, additional information and pseudo-code windows (containing HTML), input generators and "stop-and-think" questions that facilitate the *responding* category of ET. The slightly unorthodox experiment about starting programming with visualization aids, described in **publication (iv)** of this thesis has been implemented using the JHAVÉ environment.

Myller et al. [71] used visualization tools in collaborative learning and analysed visualization tools from that perspective. An empirical study showed ET to correlate with the learning outcomes also in collaborative use. The collaborative use of visualizations together with engaging visualizations has also been found efficient by Rajala et al. [80] and Korhonen et al. [48] as collaboration and discussion increase when the level of engagement increases. Also, the higher the level the engagement, the higher level of abstraction in the discussions: making the students grasp larger mental models instead of implementational details.

In a later study by Myller et al. [70] Engagement Taxonomy was extended into **Extended Engagement Taxonomy** to better explain use of visualizations also in collaborative learning. The new levels of the taxonomy, in addition to the ones in Table 3.1) are:*Controlled viewing* (viewing with controls), *Entering input* (for the program to be visualized), *Modifying* (the visualization input or program code before viewing) and *Reviewing* (for suggestions and comments, on the visualization, the program or algorithm itself).

In his doctoral thesis Juha Sorva [89] integrates the aspect of engagement from Engagement Taxonomy to the dimension of content creation, or content ownership, within the taxonomy in a new taxonomy he calls **2DET**, a two-dimensional engagement taxonomy. He states that the level *constructing* in the original Engagement Taxonomy is crowded and the order of the levels is not clear as it depends on the content that is been engaged by the student. He admits that Extended Engagement Taxonomy provides help but finds it better distinguishing between the two separate dimensions: *direct engagement* and *content ownership*. Direct engagement is concerned with the engagement the learner has with the visualization itself (very much like ET) and content ownership is concerned with an indirect form of engagement coming from the content of the visualization (*given content, own cases, modified content* and *own content*). The taxonomy is presented in Figure 3.1.

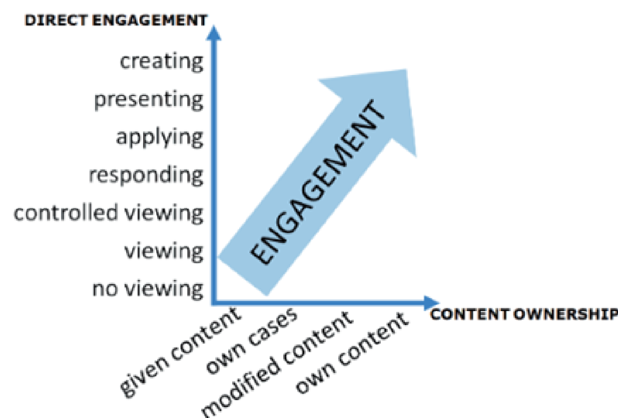


Figure 3.1: The two-dimensional engagement taxonomy, 2DET, by Sorva, taken from [89].

Altogether Engagement Taxonomy (and its extended form) have provided a great basis for the use of visualization tools to be better understood and the tools to be developed further, supporting higher levels of engagement.

3.2 Bloom's Taxonomy of Cognitive Development in CS Education

To get a tool for discussing levels of cognitive development, we will look closer into the *Bloom's Taxonomy of Cognitive Development*. It is a general taxonomy of cognitive skills, developed in the 1950's by Benjamin Bloom et al. [18]. It describes the levels of cognitive development when learning a subject. This taxonomy of cognitive domain is often referred just as the *Bloom's taxonomy* and has been rather widely used in Computer Science Education Research as a tool to discuss on different *levels of learning*, as a measurement of the depth of understanding.

Bloom's taxonomy has six categories, all building hierarchically on top of each other: 1. Knowledge, 2. Comprehension, 3. Application, 4. Analysis, 5. Synthesis, 6. Evaluation.

So, for instance in a basic programming course where *loop structures* are studied, the student having reached only the level 2, Comprehension, would understand that there are different loop structures (`for, while, do-while`) but would still struggle when writing a code using those. A student in level 3, Application, would be able to write a functioning loop structure when instructed to do so. Another student, having reached level 4, Analysis, would be already able to debug a falsely working loop-structure. In level 5, Synthesis, a student would be capable of constructing larger solutions to new problems on his own where loop structures would be used as a tool for the solution, and on level 6, he would be able to evaluate and compare the suitability and applicable use of loop structures in existing larger solutions.

Traditionally the levels are approached linearly, but the correct order and the possibility of having some of the levels laid out parallel without dependencies to all previous levels has also been discussed [38, 21]. A re-evaluation of the original taxonomy by Anderson et al. [7] even discussed whether the three top-most levels (Analysis, Synthesis and Evaluation) could be slightly parallel, and at least the last two levels would change order. The same revision also renamed the categories with verbs *(Remember, Understand, Apply, Analyze, Evaluate, Create)*. The revised version also adds a second dimension to the taxonomy, describing the type of knowledge elements: A. Factual knowledge, B. Conceptual knowledge, C. Procedural knowledge, and D. Metacognitive knowledge. The original and the revised taxonomies are shown in Figure 3.2.

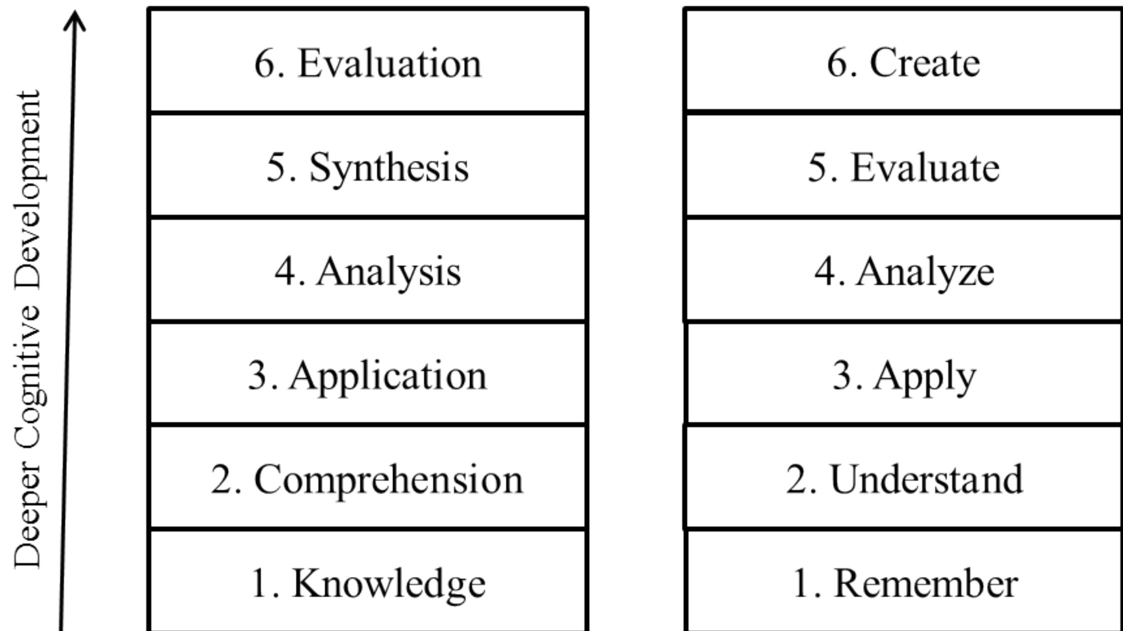


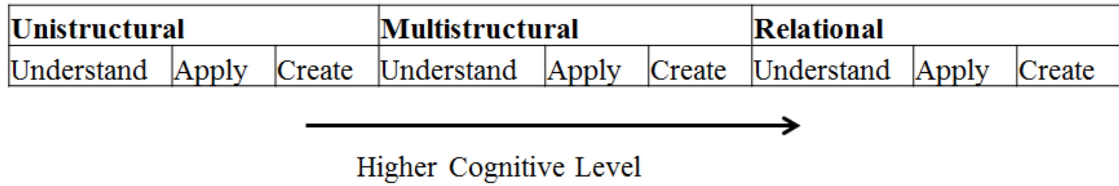
Figure 3.2: The original Bloom's taxonomy [18] and a later revision [7], without the second dimension (knowledge type).

SOLO Taxonomy (Structure of the Observed Learning Outcome) [17, Chapter 5] is a learning taxonomy that makes no reference to the cognitive level of the learner but focuses on the content of learner's response to what is being assessed. It examines the responses structural relationships within the content. Hierarchical levels of the SOLO taxonomy are **1) Prestructural** (answer is not related to topic at all), **2) Unistructural** (simple meaning, naming, focusing on one issue of a larger task), **3) Multistructural** (disorganised collection of items), **4) Relational** (understanding, using a concept that integrates the data and how to apply the concept into a familiar problem), and **5) Extended abstract** (relating to existing principle so that new problems can be solved).

SOLO taxonomy is a generic learning taxonomy but for instance Lister et al. [63] have applied it into teaching programming, to novice programmers' test answers by relating SOLO to code reading problems. They state that besides focusing teacher assessment on the first three levels of SOLO (prestructural, unistructural and multistructural), the teachers also need to test students' abilities in providing relational answers—being able *"to see the forest, not just the trees"*.

Meerbaum-Salant et al. [68] conducted a study around visual programming and wanted to have a unified, strictly hierarchical taxonomy suitable for programming tasks that would scale also for smaller programming tasks that are often used in teaching. They combined Bloom's taxonomy and SOLO taxonomy into a new taxonomy so that their taxonomy contains three super-categories defined by SOLO (unistructural, multistructural and relational) and each of these contain

sub-categories defined by Bloom's (understanding, applying and creating). The combined taxonomy is shown in Figure 3.3. Highest level in the taxonomy would be *relational creating*, and lowest level *unistructural understanding*.



Unistructural			Multistructural			Relational		
Understand	Apply	Create	Understand	Apply	Create	Understand	Apply	Create

Figure 3.3: A combination of Bloom's taxonomy and SOLO taxonomy, presented by Meerbaum-Salant et al. [68].

Bloom's taxonomy and its different revisions have been used all around CS curricula, e.g., for designing learning objectives for a course and a way to assess students. More detailed description of the use cases, especially within the CS curricula are discussed in detail in the **publication (i)** of thesis.

Despite being widely adopted, the use of a learning taxonomy is not problematic. Different teachers understand the level of abstraction of the taxonomy differently. Some teachers apply the hierarchy to individual topics as for others the hierarchy represents progress through the subject as a whole, for example in a degree programme [38]. Individual teachers also have different opinions in how to place individual programming skills into the taxonomy [38]. Lister and Leaney [62] also state that based on the size of the assignment, a code writing task, for instance, can belong to various levels of Bloom's taxonomy.

Lahtinen [52] studied the correspondence of Bloom's taxonomy to programming students' skills by separately testing different levels of the taxonomy. In her study, she formulated a test for a programming course of 254 students that would hold questions ideally measuring a certain Bloom's taxonomy level of one topic. The study presents a statistical cluster analysis on how students had reached different cognitive levels of an individual subject. As a conclusion, she found out that the students' cognitive skills were not in all cases following the linear hierarchy of Bloom's. Instead, she found new kinds of student categories, each with a skill-set of their own like *theoretical, practical* or *memorizing students*. For instance the *theoretical* students had obtained high theoretical understanding of the subject but were incapable of producing their own code or applying their information to relevant practical problems where as *practical* students were able to write program code without the ability to analyse or evaluate even their own program code.

Partly based on this finding, **publication (i)** of this thesis presents a revised Bloom's taxonomy to better match Computer Science Education, now splitting it into a two-dimensional matrix by separating practical and theoretical skills in dif-

ferent axes. Both **publications (i)** and **(ii)** of this thesis discuss the application of Bloom's taxonomy into the usage of visualization systems to support deeper levels of cognitive development through improved engagement. This does not only introduce direct benefit of learning by the tool but also facilitates the consistent integration of visualization systems into the course setup which then itself improves the effectiveness of visualizations [55].

3.3 Summary

Learning taxonomies can be used as a tool for a teacher to define assessment strategies, examples and assignments that support deeper levels of cognitive development. **Publication (i)** of this thesis discusses the suitability and applications of different learning taxonomies to Computer Science Education in general and suggests a two-dimensional adaptation of Bloom's taxonomy to better explain the speciality of learning programming: reading and writing code are two separate skills that do not necessarily require each other.

In the case of visualization examples, examples that support deeper cognitive development require enhanced engagement as suggested by the Engagement Taxonomy and the adapted Bloom's taxonomy presented in **publication (ii)** of this thesis. Usage of these more engaging visualization examples does not only emphasize deeper learning but also allows better integration to the structure of the course as visualizations can be a continuous part of the learning materials. **Publication (iii)** describes a practical example and empirical findings of such an approach and **publication (iv)** presents an example of integrating engaging visualizations to a course even without having taught any programming yet.

4. TOOLS-ASSISTED ASSESSMENT IN PROGRAMMING COURSES

A big part of organizing successful programming courses is a successful assessment strategy. As learning programming requires *doing* programming, the teacher needs to have the students create enough of their own programs, assess and provide feedback of them but also maintain realistic workload for himself. In this chapter we will look into how all this can be facilitated by different tools.

We will first start by looking into the assessment of programming courses in general and continue with an overview to computer-assisted assessment tools that can help by assessing student programs automatically. Then we will look into a semi-automated assessment process for programming courses which combines both automatic and manual parts. Finally, we will end with an introduction to rubric-based assessment tools that then facilitate the manual assessment part.

4.1 Assessment and Programming

We will begin by looking into defining the different aspects and functions of assessment, especially in the context of assessing programming assignments. On one hand, the assessed items of a programming course, the programs, are very outcome-based: the program works or does not work as specified (functionality). On the other hand, the *process* of designing, developing, testing and debugging, and the best practices for doing so, are important to be part of the assessment as well.

According to Gibbs [29] assessment has six main functions:

1. Capturing student time and attention.

2. Generating appropriate student learning activity.

3. Providing timely feedback which students pay attention to.

4. Helping students to internalize the discipline's standards and notions of quality.

5. Marking: generating marks or grades which distinguish between students or which enable pass/fail decisions to be made.

6. Quality assurance: providing evidence for others outside the course (such as external examiners) to enable them to judge the appropriateness of standards on the course.

Functions 3 and 4 are forms of *formative assessment* which is used to determine whether a learner has reached sufficient level of knowledge on a subject before the chance to learn the subject has passed. Based on formative feedback, the student can then guide his focus and efforts towards the final learning goals of the course. It is important not only for the student to know his level and what is expected of him, but also for the teacher to recognize possible misconceptions and adjust the teaching to address them [4].

Functions 5 and 6 are then forms of *summative assessment* which is carried out to determine the level of the work or knowledge to be able to define a grade, usually taking place at the end of a module or a whole course [4]. These functions are more expensive to perform but take only take place less frequent than formative assessment [29].

Sometimes final grades can be given based on a norm-referenced grading scheme, based on a statistical distribution. Lister and Leaney [61] state that this does not guarantee objective assessment in relation to course objectives. Especially in programming courses where adjacent courses are strongly based on the knowledge achieved in the previous courses. The weaker students can pass without actually being guaranteed to have sufficient competency and the best students are not challenged enough. As a solution, Lister and Leaney suggest that a criterion-based grading scheme works much better for programming courses than a traditional norm-referenced scheme. In pure criteria-referencing, explicit clear criteria for each grade are communicated to students so they can select prioritize and focus their efforts: For lower grades, the students are required to complete tasks that require them to demonstrate only lower levels of cognition (ability to read and understand programs) where as stronger students, aiming for best grades are challenged with open-ended tasks requiring synthesis and evaluation skills [62].

Having clear and transparent grading criteria is related to functions 1 and 2 in the above-mentioned list. These functions should be supported by both formative and summative assessment. Gibbs [29] found out that students spend very little of their time outside class for learning tasks that are not assessed and on the other hand took continuous formative (and inexpensive to organize) assessment as a clear learning situation. Gibbs states that, *"If it's going to have a profound influence on what, how and how long students study then it might as well be designed to have educationally sound and positive influences rather than leaving the consequences to chance."* One way for effectively increasing the amount of formative assessment for the students is to utilize peer assessment or self-assessment [4].

Proper assessment of computer programs consists of the actual application but also on its design and the process for creation the application. Assessing the actual application is very outcome-based: Does the application do what it is supposed to do? Is it functionally correct, and, if applicable, does it perform its function in given time and resource constraints. The latter criteria are usually paid more attention in more advanced courses, for instance with algorithms.

Assessing program design is something that can be approached in different levels based on the focus of the course but especially on the used programming paradigm. In an object-oriented programming course, proper design of objects and their interaction is usually one of the main assessment criteria where as in an introductory programming course teaching imperative programming structures the syntactical and semantical implementation details and program-structure in smaller granularity are weighted more.

Especially in the introductory programming courses, it is necessary to pay attention to the programming process as well. Besides programming concepts, the students are taught good programming processes and best practices. This contains topics like "design before implementing", "proper testing, and planning of tests" and "debugging strategies". Even though the process could be documented by the students and the document then assessed, this would not probably lead into objective assessment of the real process, or thus guiding the students to implement a proper process either. This can be guided and assessed by requiring the students to return design documents before the implementation phase, making them design and submit their own tests for the program [25] and making the whole process incremental with multiple return phases [1].

Another important aspect in assessing student-submitted programs is sufficient and personalized feedback. The same result, a properly-functioning program, can be reached in various different ways—some of them better than the others. In order to best guide the student forward, the teacher needs to study his program code and write personalized feedback that resembles that solution that the student has taken. This naturally takes a lot of time and effort.

In all, we can summarize the following about assessment and programming courses:

- Assessment, divided into formative and summative, serves multiple functions and will guide the learning activities and time allocation of the student.

- Sufficient amount of formative assessment is useful to guide the students and the instructor and relatively cheap to organize, for instance using self-assessment or peer-review.

- Assessment of computer programs requires looking into multiple aspects: functionality, program design and the whole development process.

- As same functional program can be reached by different ways the assessment is personal and unique to some extend and good feedback is then personalized to match the choices the student made.

4.2 Computer-Assisted Assessment

Testing the functionality or other features of a computer program using another computer program is a very natural thing to do and thus not a very new idea: Automated testing tools have existed all-along in professional usage of software industry for various testing, validation, and verification purposes. As assessment (a form of verification) is a vital part of programming education, it is natural that there are systems available for computer-assisted assessment (CAA) in education as well. In this section, we will look into how the usage of CAA systems can facilitate the work of a teacher and enhance the learning experience of the student in a programming course.

The most common purpose for a CAA system is to check the correct functionality of student program [22], but there are multiple parts that can be assessed automatically by a CAA system. Ala-Mutka [6] provides a wide survey of CAA-systems in use before 2005 and presents a division of features that can be assessed by a CAA system. The division is presented in Figure 4.1. The CAA systems can perform both *dynamic assessment* (run-time tests for a compiled application) and *static assessment* (analysing the program source code).

Ala-Mutka [6] divides Dynamic Assessment into measuring *functionality* (the behaviour of the application, for instance, using given test input set and comparing the result into model solution output), *efficiency* (measuring CPU time, dynamic profiling of efficiency, *testing skills* (making sure students test their own programs with their own test sets that are assessed, test coverage) and *special features* (e.g. language specific features like dynamic memory allocation and memory leaks). The static assessment is then divided into *coding style* (good programming practices, commenting, language specific features that are for instance forbidden within the course), *programming errors* (recognizing suspicious code fragments through static analysis, recognizing typical coding errors, code redundancy), *software metrics* (general measurements about the program such as complexity, lines of code, often related to coding style of good practices), *design* (use of certain structural choices or a given interfaced) and *special features* (eg. plagiarism).

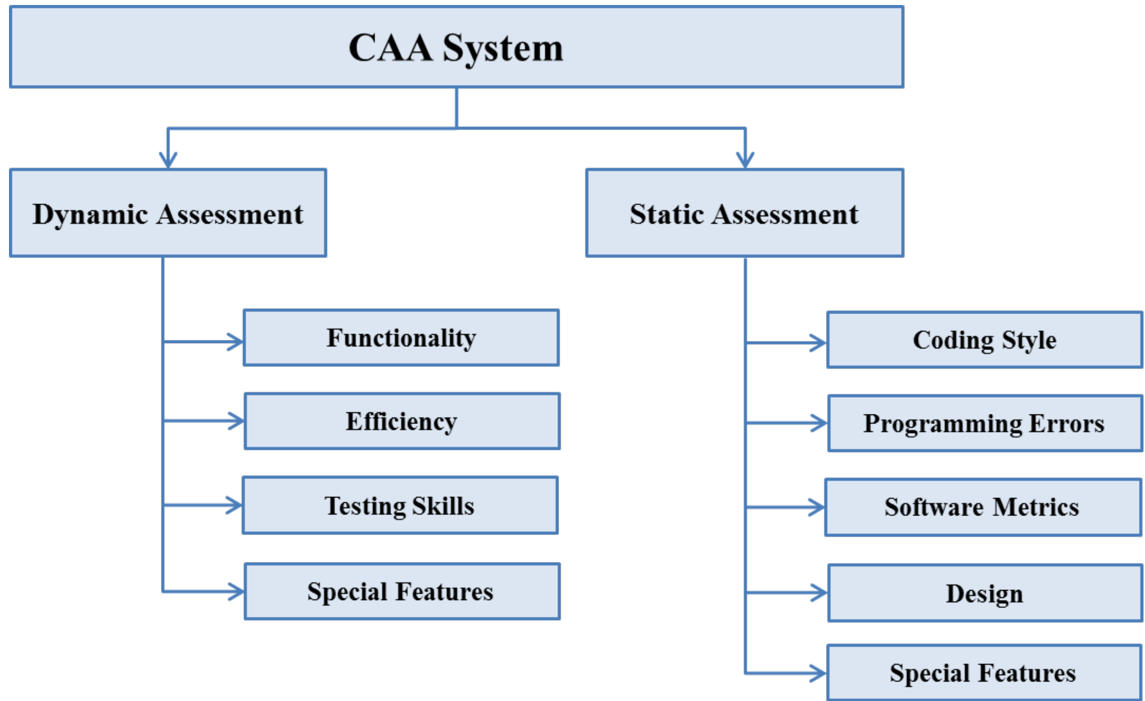


Figure 4.1: Features assessable by a CAA systems, gathered by Ala-Mutka [6].

Douce et al. [22] present a wide literature review on the history of CAA tools both from the perspective of evolution and from their pedagogical aspects. They divide the evolution of CAA tools into three generations:

- **First Generation — Early Assessment Systems.** Starting all the way from 1960's automated assessment systems have existed as long as educators have asked students to build their own software. The early systems were implemented on testing punch cards. The early systems were focusing around measuring program size, execution time and comparing test results to given data sets.

- **Second Generation — Tool-Oriented Systems.** These systems were heavily based on efficient use of existing operating system tool sets that were chained either from command-line or from separate graphical user interfaces. Many of these systems are based on character-by-character or regular expression comparison of program output compared to expected output. The systems could be used directly by students locally or at least within the same UNIX environment, without remote client-server access to submit and test their applications. A lot of effort was then required to ensure that the tested student application could not execute harmful code or that the test input data was not accessible or visible for the student. Some examples of these tools are **ASSYST** [37], **Ceilidh** [16] and **BOSS** [39].

- **Third Generation — Web-Oriented Systems.** The third generation of tools makes use of development in web technology providing an easier way for students to submit their applications through their web browsers, receive feedback and the teachers/administrators to manage the submission system set-up in general. The testing features themselves have evolved into more sophisticated formats. Examples of these systems include **Course Marker** [31] (a successor of Ceilidh, previously called Course Master), later evolutions of the the **BOSS** system [39], and **Web-CAT** [27] which also emphasizes test-driven development by letting students submit their own tests to accompany their program [25].

Carter et al. [21] conducted a wide international study for CS educators in 2003 to find out about their usage and attitudes towards CAA. In general the perception towards CAA was positive: the respondents agreed strongly that CAA reduces marking time and improves the immediacy of the feedback and to some extend agreed that CAA improves objectivity of the grading and flexibility for the student. When looking at the negative aspects, system failures, downtime, data corruption and plagiarism had widespread concern. They also found out that teachers who were inexperienced in using CAA found the systems to be too limited to measure higher-order learning outcomes and that the quality of the immediate feedback was poor. However, these negative assumptions diminished as a respondent's experience on CAA improved.

In **publication (v)** of this thesis we will look closer into the more recent evolution of CAA systems and their features, providing a wide survey of CAA tools from 2005 to 2010. The work is an updated increment of the earlier survey done by Ala-Mutka [6], cited in the beginning of this section.

4.3 Semi-Automatic Assessment

As effective and valuable as computer-assisted assessment can be, it is not something that can be taken into use without proper integration to the course logistics and it is not ideal that CAA systems would completely obsolete the importance of human assessment, as many of the most important assessable features of a program are the most difficult to find out with only CAA [36]. As a compromise Ala-Mutka and Järvinen [5] have defined a semi-automatic process for efficient assessment of programming courses that has been adopted in the programming courses of Tampere University of Technology. The process integrates both computer-assisted assessment and manual assessment to best facilitate good grading, proper and accessible feedback in a programming course with realistically limited resources for a large amount

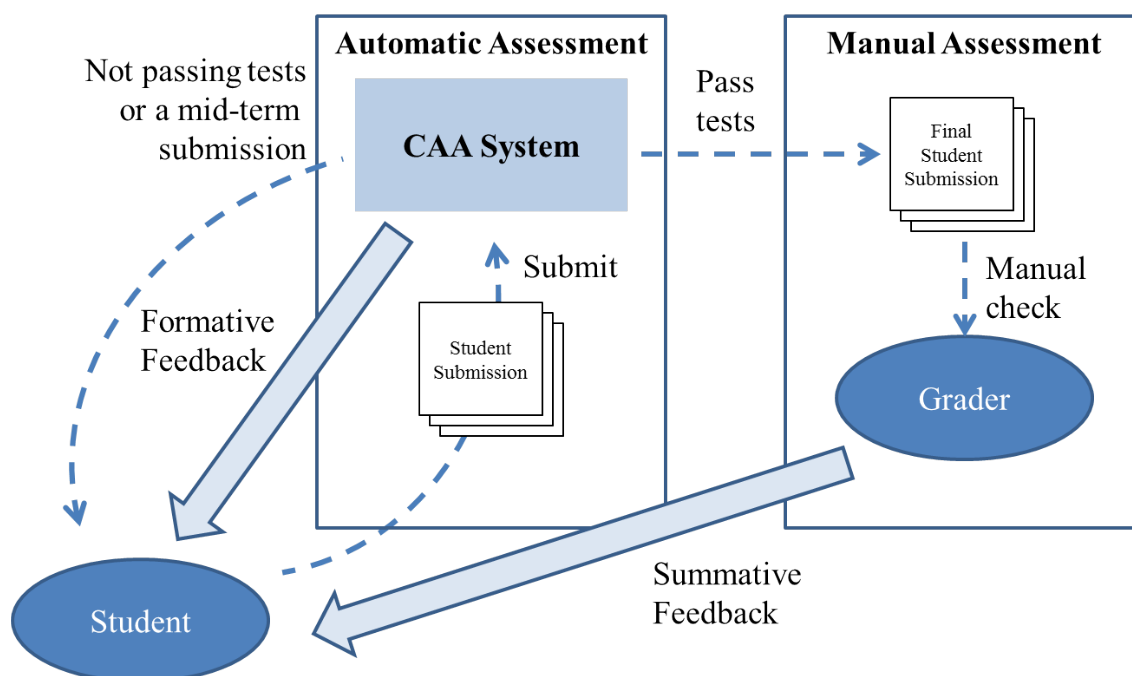


Figure 4.2: Process of Semi-Automatic Assessment.

of students. This section will introduce the process and is based on the article mentioned above.

The process for semi-automatic assessment is described in Figure 4.2. The student works with his program code and can directly use the Computer-Assisted Assessment system to submit his work for benchmarking. This can be purely a formative submission if an incremental assignment with multiple phases is used. Then at least for the final submission phase, the student submits his work to see if he passes a certain minimum level required by the automated tests. These tests run dynamic tests for functionality but can also contain static code analysis in the form of a style or test coverage analyser. Nevertheless, from the system, the student receives formative feedback to know how his program is working against those tests.

When the student program is ready for the final submission, for instance passing the sufficient limit for functionality, the final assessment is done manually by grader (teacher or teaching assistant). Now, already knowing the student code functions properly, and seeing the results from the CAA tests, he can focus his time on the items that were not tested automatically. These include programming style, structural design and best practices in programming, whatever the focus of the course is. During his final grading, the grader naturally writes feedback for the student and marks the final grade for the work.

Even though the semi-automatic assessment approach is usually chosen because one has to comply with limited resources, it introduces a lot of positive aspects to the grading compared to using only the CAA system or grading everything manually:

- The student can receive immediate formative feedback from the CAA system regardless of the day and hour, without needing to burden the teaching staff.
- The grader does not need to manually test the functionality or other automatically assessable features of the application but he can focus on the deeper aspects of the student submission.
- The student work is properly assessed by a real person, providing personal feedback that will guide the student towards better programming practices, not just better-functioning applications.

What is left out from Figure 4.2 but addressed by Ala-Mutka and Järvinen [5] is that the student is naturally supported also during the programming process by not being left alone to work with computer and definitions. The process includes contact educational situations, tutor appointment hours and electronic discussion groups with a list of frequently asked questions on the web page.

4.4 Rubric-Based Assessment Tools

Semi-automatic assessment process integrates best of both worlds from computer-assisted assessment and manual assessment. Manual assessment can too be assisted using tools. This section introduces rubric-based assessment tools that facilitate the manual assessment labour, especially when multiple graders are needed and thus possible problems of objectivity and inter-grader consistency are introduced.

Introductory programming courses in Tampere University of Technology (TUT) are set-up as mass courses with up to multiple hundreds of students. As programming is best learnt by programming that means a lot of program code is being reproduced in these courses. It is easy to see that one teacher cannot properly handle the amount of student program code written in the course so that the students would get any valuable feedback from their work. Teaching Assistants (TAs), that are for instance senior or post-graduate students, can be used to make the burden easier. In introductory programming courses in TUT it is common to have over TAs assessing the student programs of one course.

When multiple graders are used for assessment **objectivity** and inter-grader **consistency** might become issues. The teacher needs to make sure each grade is given in a consistent way regardless of which of the graders has given the mark, so that for the student it is irrelevant who of the TAs will end up marking his work. Also, the graders need to be objective and follow the grading scheme defined by the teacher consistently. The grader must not be biased on who is the (possible fellow junior) student. Also, if the student has done really overwhelmingly lousy job on one part of the work but the rest of the work is decent, he is still entitled to the points of the other parts he deserves based on the objective grading scheme.

PROGRAMMING STYLE	0	1	2
Amount of comments	No comments	Some comments here and there	Consistent and descriptive that guide the reader of the code
Naming of variables and functions	Undescriptive and irrelevant, *(like 'a', 'b', 'variable', 'foo()')*	Some good, some bad	Consistently descriptive *(like 'cumulativeSum', 'initializeDataStructures()')*
Forbidden features	goto used	Macros used instead of const variables	Everything OK

Figure 4.3: An example rubric, showing some smaller pieces that could be part of assessing "Programming Style". The grader goes through each row (criteria) selecting the matching requirements. The sum of the points is then used to formulate the full grade.

According to Habeshaw et al. [30], the only definite way to grade objectively is by using only objective tests like multiple-choice questions. This is not suitable in courses where a major part of the learning is based on the student programming by himself and the assessable item is the program written by the student. As a suggestion, Habeshaw et al. state that in order to ensure consistency among graders, the graders need to be properly trained for the job and are required to use marking schemes to direct their attention to the appropriate things in a student's work. A a solution for grading programs, Becker [12] suggest creating relatively detailed rubrics that outline what students must do to meet and exceed the requirements.

A **rubric** is a two-dimensional grid where each row describes one part of the assessment, an element of the program, problem or solution, and each column relates to a level of achievement [12]. The idea is to slice down the assessment of a large program into small enough pieces that can be graded by anyone (with sufficient training) completely objectively by just deciding what requirements (column) the submission meets in the selected row (criterion).

An example part of a rubric, displaying for instance how *"Programming Style"* could be split into smaller pieces, is shown in Figure 4.3. The grader goes through each row, that matches an individual feature to be looked and selects the column with the criterion that the student's program code meets. The final grade is then formulated based on the points gathered from each of the row. The grade can be just a range of the cumulative points gathered from the rows or a differently weighted calculation of the different parts. For instance in a situation where it has been decided (and maybe even communicated to the students) that "Programming Style" forms 30 per cent of the final grade.

The rubrics can be used in a traditional paper format or using a computer-based grading rubric. Paper format rubrics are not helping the grader to write the feedback. Even though marking would be easy, the grader would still need to create

the feedback manually to the student. This is something that computer-assisted rubrics can facilitate easily. When marking one of the rows of the rubric, the tool can generate a corresponding feedback phrase that can then be customized quickly to match the exact issue the grader wants to point out. This is a feature we call **semi-automatic phrasing**.

Anglin et al [8] conducted an empirical study about the efficiency and effectiveness of using computer-based rubrics compared to other grading methods. Results suggested that using computer-assisted rubrics were almost 200 % faster than traditional hand-writing without rubrics, more than 300 % faster than traditional hand-writing with rubrics, and nearly 350 % faster than typing the feedback manually into a Learning Management System. Manual rubrics were slower than manual assessment without rubrics mostly because it took time to create the rubric, so reusing the rubric would fade out that difference. The effectiveness of the feedback (how students felt about the quality of the feedback) did not hinder from the use of a computer-assisted rubric tool. Similar positive findings about student satisfaction towards feedback and decreased time used for grading were also found by Atkinson and Lim [9] in their empirical experiment, where they used rubrics that were integrated into their learning management system [1].

Edmison et al [24] raise the importance of *contextualized feedback* in conjunction with rubrics-based tools. They state that even though many of the computer-assisted rubric tools increase consistency, objectivity and help in generating customized feedback, the feedback is taken out of context from the original student submission. Especially in some cases, it would be important for the marker to be able to add comments directly to the context. They present an extension plugin for Moodle that allows creating rubrics but also creating contextual feedback.

Publications (vi) and **(vii)** of this thesis present the idea and benefits of using a computer-assisted rubric tool. The publications present empirical studies around the effectiveness of the tool's usage on programming assignment looking into objectivity, consistency, time usage and time distribution of teaching assistants along with the quality of the feedback written using *semi-automated phrasing*. The example tool that was used is called **ALOHA** [3], which was used in Tampere University of Technology from 2005 to 2009. Later, a successor tool called **Rubyric** [10] was developed based on the findings of these above-mentioned articles.

4.5 Summary

Assessment is a versatile and important part of the learning process both for the student and for the teacher. Assessing computer programs has its own twists as

[1] They used BlackBoard 9.1 with Blackboard Rubrics, see http://www.blackboard.com

there are a lot of features, both dynamic and static that need to be taken into account. On the other hand, the programming courses in general are organized as mass courses with limited human resources. Luckily, big part of assessment can be facilitated by automating parts of the assessment.

Using computer-assisted tools in the assessment of computer programs have become a natural part of programming courses and such tools have existed in research also for the past 50 years. There are plenty of tools available for the teacher to select the most suitable one for his approach and course setup. A common open format for assignments has been suggested by Edwards et al. [26] but in general the CAA tools are very independent of each other.

Even though many things can be automatically assessed by a CAA system, it is important to manually assess the student programs as well. Semi-automatic assessment provides a good model for taking the best of both worlds. Especially when using multiple graders for the manual assessment, grading rubrics and rubric-based tools provide a valuable asset for the teacher to retain the grading objective, consistent and provide feedback of good quality.

5. SUMMARY OF THE INCLUDED PUBLICATIONS

In this chapter, each of the publication is shortly introduced and summarized to form the big picture of the whole study. The next chapter then compares the results to the research questions of the thesis.

Publication (i) presents a review on different learning taxonomies that are used within Computer Science Education and a new adaptation of Bloom's taxonomy to better fit Computer Science Education.

Publications (ii), (iii) and (iv) discuss the different ways of using Program Visualizations (PV) on a programming course. Publication (ii) provides a more theoretical approach where as (iii) and (iv) introduce practical applications and evaluation on the used approaches.

The rest of the publications, (v), (vi) and (vii) focus on the process of using semi-automatic assessment in programming courses. Publication (v) provides a literature review on the set of existing automatic-assessment tools. Publications (vi) and (vii) then introduce and evaluate rubric-based assessment tools.

Publication (i), *Developing a Computer Science-Specific Learning Taxonomy*, presents a literature review of learning taxonomies especially from their adaptivity to Computer Science Education. Additionally, a CS-specific adaptation of Bloom's Taxonomy of Cognitive Domain is introduced. As the author of this thesis was part of the group that developed that adaptation, we will focus on Bloom's taxonomy and the new adaptation.

Bloom's Taxonomy of Cognitive Domain is explained in more detail in section 3.2 of this thesis. The traditional taxonomy by Bloom et al. [18] and it's later adaptation [7] place the six levels of cognitive development mainly in a hierarchy depending on each other. **Publication (ii)** presents a categorization of visualization examples built to correspond the different levels of Bloom's taxonomy.

What has been discovered when adapting Bloom's taxonomy into practical use in teaching programming, for instance by Lahtinen [52], is that reading, or *interpreting*, and writing, or *producing* program code are two separate skills and not necessarily dependant on each other. Based on these findings, the CS-specific adaptation of Bloom's taxonomy, namely **the Matrix Taxonomy**, splits the levels of Bloom's into

PRODUCING
Create
Apply
none
Remember
Understand
Analyse
Evaluate
INTERPRETING

Figure 5.1: A graphical presentation of the two-dimensional, CS-specific adaptation of Bloom's taxonomy, the Matrix Taxonomy, taken from **publication (i)**.

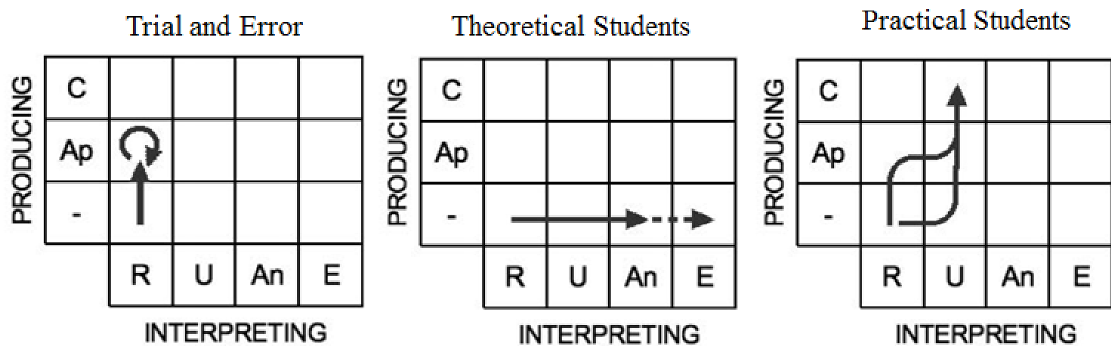


Figure 5.2: Undesirable learning paths the students can take when learning programming: Trial and Error approach, Theoretical Students and Practical Students, taken from **publication (i)**.

two dimensions, separating theoretical skills (reading) from practical skills (writing). The result is graphically presented in Figure 5.1.

The idea of the two-dimensional Matrix Taxonomy is that it can now be used to describe not only levels of student's knowledge on a certain subject but also the paths, or learning strategies, the different students take when learning programming. Using the student groups found in the cluster analysis of students' programming skills by Lahtinen [52], we can for instance identify that *Theoretical students* have attained only levels Understand and Analyse without being able to Apply their knowledge on a simple problem themselves. Figure 5.2 shows some of these wrong learning paths students sometimes take.

The contribution of this new adapted taxonomy is to provide tools and vocabulary to better describe and communicate the levels of knowledge the students acquire and teachers to be able to pursue them towards the top-right cell, Create/Evaluate,

which could also be recognized as the level *Higher Application* of the Bloom's revision defined by Johnson and Fuller [38]. One example of later practical usage for the Matrix Taxonomy is a defined process for setting up a programming course on embedded programming where students are guided to take the correct learning paths, defined by Vanhatupa et al. [95]

Another suggestion of the publication is that as *abstraction* is an important skill applied all around programming, the more advanced programming skills also build on top of the previous skills. This means that these learning taxonomies can be applied iteratively going from abstraction layer to another. Similarly, it implies that if a student has not reached sufficient knowledge on a basic skill, or has misconceptions around it, it will certainly cause problems when more abstract, higher-level topics are taught. Fixing the situation might require returning into the previous topics. Teachers can use these taxonomies as a tool to make sure their learning goals are sufficient, the goals are communicated sufficiently, example materials reflect and enhance deeper levels of knowledge and that the teachers are assessing the right matters. As discussed in Chapter 4, assessment directly guides students' focus and eventually their learning.

Publication (i) has received quite a lot of attention in the past years within the research field. According to Google Scholar [85] statistics, the publication has been cited 100 times during 2007–2014.

Publication (ii), *Visualizations to Support Programming on Different Levels of Cognitive Development*, introduces a categorization of program visualization examples based on the Bloom's Taxonomy of Cognitive Domain. The starting point for the research was the urge to expand the use of program visualizations in introductory programming courses. Mostly program visualization examples were used with no or only minor possibilities for student interaction—as sort of *click-and-watch examples*. There is nothing wrong with these kind of examples, they are a very effortless way for a teacher to present a new topic or for a student to self-study, e.g., a missed lecture.

As has been concluded in the publication, without interaction these *click-and-watch* visualization examples only support shallow learning, the first levels of Bloom's taxonomy. Thus, they have been placed on category *illustrative visualizations*. This also limits the possibilities of using those examples effectively throughout the whole course. If the visualization tool can be used all-around the course and not just here and there when a new topic is introduced, the students adopt the tool better and find it more beneficial [51].

Table 5.1: Program visualization examples categories to correspond Bloom's taxonomy.

Bloom's taxonomy level	**Visualization category**
1. Knowledge	-
2. Comprehension	Illustrative visualizations
3. Application	Utilizing visualizations
4. Analysis	Problem-solving visualizations
5. Synthesis	Productive visualizations
6. Evaluation	Discerning visualizations

To support deeper learning and thus reach more use cases in a programming course the examples need to address other levels of Bloom's taxonomy with more engagement. For this, Naps et al [73] explored sample tasks and assignments for *algorithm visualizations.* Publication (ii) introduces new categories for program visualization examples, presented here in Table 5.1.

Briefly the idea behind the categories is that, for example, a student, who is studying loop structures would be able to practice creating simple loop structures with an *utilizing visualization* thus reaching level 3, Application, in Bloom's taxonomy. With successfully practising with *problem-solving visualizations* student could progress to level 4, Analysis. This could be in practice done for instance with a *debugging visualization*, a subcategory of problem-solving visualization, where the student needs to analyse and fix a broken loop structure.

To assure all the different new visualization categories are actually achievable, the publication presents practical ideas on how to build these visualizations on each level. At the time of the publication the PV tools did not really support the new categories so the publication concentrates on the ideal level, providing more theoretical input for the people responsible for creating the PV tools. This was done later for instance with the VIP tool [54].

According to Google Scholar [85] statistics, publication (ii) has been cited 13 times during 2005 and 2014.

Publication (iii), *Visualizations in Preparing for Programming Exercise Sessions*, is a rather direct continuum for publication (ii). It concentrates on two matters:

1. How to support integration of program visualizations into an introductory programming course through exercise sessions?

2. Evaluation of the effectiveness of such approach.

The publication presents a research experiment that took place on an introductory (CS1 level) programming course in Tampere University of Technology. As in earlier

years of the course, the students had *illustrative visualizations* for their use in the course web pages. These they could use for self-studying a topic and these were also occasionally presented in the lectures. However, these were used by only a handful of students throughout the whole course, and they acted more as an extra material for the most active students.

Besides lectures, the course contained weekly exercise sessions. For these the students were required to review the lectures notes of the topic and implement a very small homework assignment with pen and paper which they were to return for the teaching assistant in the beginning of the session.

This year, we introduced a visualization-based alternative. Few student groups got to try *(or actually, were forced to try)* the preparation for the exercises with the VIP visualization tool [97]. The students had *illustrative visualizations* for reviewing the lecture notes and then an *utilizing visualization* and a *problem-solving visualization* for the homework assignment. So, the students had visualization tool support also for deeper levels of the Bloom's taxonomy with engagement involved.

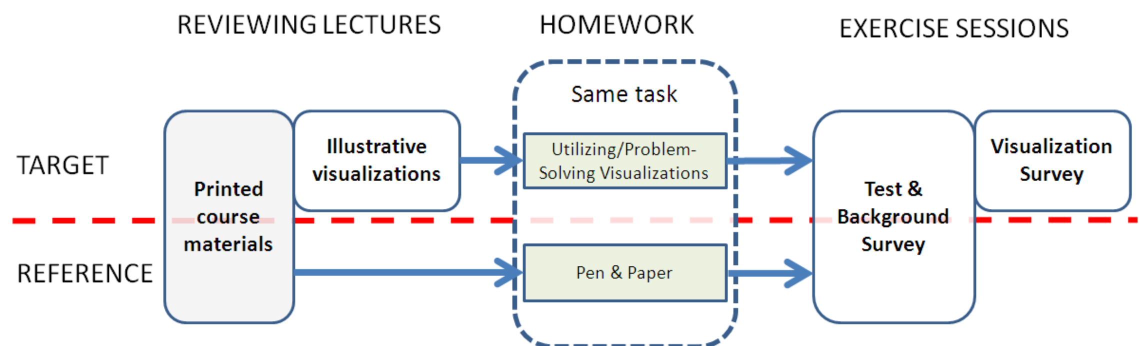


Figure 5.3: Organization of the experiment in publication (iii).

For evaluating the effectiveness of our approach we used a target group (using visualizations) and a reference group (traditional approach with pen and paper). We evaluated two measures: effects on *learning results* (how well were they prepared for the topic) and effects on *studying behaviour* (how did they prepare).

To study the learning effect we had a small test in the beginning of the exercise sessions of which the students did not know beforehand. For the studying behaviour we conducted a survey, mostly concentrating on how much the students spent time for the preparation. The target group also had an extra survey about the use of visualizations. The whole organization of the experiment is presented in Figure 5.3. The experiment was done twice, in adjacent weeks.

The study showed that visualizations were statistically improving the students' learning outcomes if the student was new to the topic or having difficulties in the course. This background information was collected in the survey. If we compare all

students in the groups, the difference is not statistically relevant, but the students who already know the topic are not really the target audience for this approach.

Another positive outcome of the study rose from the survey: Visualizations had a huge impact on the studying behaviour of the students. The target group used much more time on the preparation.

With the two results combined (better learning outcomes, more time spent on preparation) one can clearly conclude that the approach was well beneficial. However, it can not be concluded that *"Students did learn better when using visualizations because they would be pedagogically more efficient"* as the students did spend more time on preparation.

What can be concluded on the basis of the study is that *"Students did learn better when using visualizations, either because of the effectiveness of visualizations or because the students just spent more time studying with them."* Both of the reasons were supported by the open answers in the visualization survey—students enjoyed using the tool but also felt they could understand the topic better with the tool.

According to Google Scholar [85] statistics, publication (iii) has been cited 11 times during 2007–2014.

Publication (iv), *Kick-Start Activation to Novice Programmers - A Visualization-Based Approach*, introduces an experiment where a visualization tool was used in the very first lecture of the first introductory programming course, to present *what is programming*. The publication itself is not only on visualizations but on activating novice programmers with their misconceptions. From the viewpoint of this thesis, the publication presents a novel way of using engaging visualizations.

The publication presents an approach, a different way to start an introductory programming course named *a kick-start activation*. In this approach the deep structure of programming is presented with visualizing algorithms in flow-charts and pseudo-code before the surface structure is at all touched upon. The idea is to directly "throw the students to the deep end" with the assistance of a visualization tool.

In this study, it was impossible to use a traditional program visualization tool as these work on program code level and now the students knew nothing about any existing program code. Algorithm animations, on the other hand, quite often work without any existing program code as algorithms are expressed for instance with pseudo-code. So, an algorithm animation tool called JHAVE [72] was then taken into use and modified so that it would animate the example kick-start algorithm (hyphenating Finnish words). The algorithm itself was presented with a pseudo-code and a flow-chart. The visualization was then used in the lectures and the

students had the possibility for accessing it during/after the lecture.

As the visualization example resembled more of an algorithm animation than a program visualization, the Engagement Taxonomy (see section 3.1) could have been applied for improving the example. The example was controllable by the students which makes the example reach level *Change* of the Engagement Taxonomy. To attain the level *Response* also, the flow of the program was interrupted with pop-up questions querying about the next behaviour of the algorithm.

A small evaluation was performed with a quantitative survey after the lecture where it was used. In general, the students found the visualization based approach useful and somewhat interesting. The most important positive finding was that especially the students who had no previous programming experience found the approach useful for learning.

According to Google Scholar [85] statistics, publication (iv) has been cited 7 times during 2009–2014.

Publication (v), *Review of Recent Systems for Automatic Assessment of Programming Assignments*, is a survey and a systematic literature review of the recent (2005–2010) development of computer-assisted assessment (CAA) tools for programming exercises. Idea of the publication was to update the previous large study done in 2005, *A Survey of Automated Assessment Approaches for Programming Assignments* by Kirsti Ala-Mutka [6].

Even though some literature reviews had been conducted since Ala-Mutka's study a systematic collection of trends and improvements in the rapidly evolving area of CAA tools was not done. For that gap, publication (iv) addresses the following research questions:

1. What are the features of CAA systems reported in the literature during 2005–2010?

2. What future directions are indicated?

To answer the questions, a large amount of publications about CAA of programming assignments were collected and studied from *ACM Digital Library* and *IEEE Xplore*. These publications were then read and categorized in order to get two results:

- A categorization of different major features that CAA tools support.

- List of those CAA tools (that have been lately introduced for academia).[1]

[1] Old and the most known tools were not focused here as the research was a continuum study.

Table 5.2: Major features where CAA tools differentiate from each other, described in **publication (v)**.

Feature category	**Examples**
Variety of programming languages	One language (Java, C++/C, etc.) vs. language independency
Integration to an existing LMS	Integrating to Moodle mostly
Ways to define tests for student code	Using industrial solutions like XCode, spesialized solutions like output comparison or scripting
Resubmission policies	Limiting the # or submissions, time penalties, etc.
Possibility for manual assessment	Teacher can change the grade manually
Sandboxing	Different security solutions
Distribution and availability	In-house vs. Open Source
Other, more special features.	GUI testing, concurrency, SQL, Web Applications, etc.

Table 5.2 presents the major features which the CAA tools support in different ways. Detailed explanations and the different observed possibilities for the categories are explained in detail in the publication.

Purpose of the categorization is to provide practical input for two parties:

- **Teachers** can use the list of features for comparing different tools when he/she needs to select a suitable tool for a programming course.

- **Tool implementers** can use the list for finding possible solutions to problems they would want to solve with their tools.

For the future directions, the publication concluded to expect more novel research to emerge from the following trends: Integrating CAA into LMSs (Learning Management Systems), Security issues of CAA and CAA of Web Applications. What is also emphasized in the publication is the possibility of using existing open source systems like Web-CAT [27], which is customizable and extensible with possible new features the teacher might need. The situation seems not to have changed since previous studies: the CAA tools are usually developed in-house to suit just the one purpose and the wheel gets re-invented here and there all the time.

Publication (v) has also received a lot of attention within the research field. According to Google Scholar [85] statistics, the publication has been cited 80 times during 2010–2014.

Figure 5.4: Grading view in ALOHA, *the rubric.*

Publication (vi), *Improving Pedagogical Feedback and Objective Grading*, introduces a concept of a *rubric-based assessment tool* in the form of a tool called ALOHA. It is an online grading tool that is originally designed for facilitating problems in the grading process of programming courses, especially mass programming courses.

If grading is done using semi-automatic assessment (see Section 4.3) CAA tools like the ones listed in publication (v) are used to facilitate some parts of the grading process. However, the workload for the manual assessment part can also be rather overwhelming, especially when there are hundreds of student assignments. In this case, multiple graders (for instance Teaching Assistants (TA)) are used.

Using multiple graders for the same assignment may cause problems in objectivity and consistency. This can be avoided by using unified grading schemes amongst graders, or as Becker defines these: *rubrics* [12]. A rubric divides the formation of the grade into assessing small enough sub-parts out of which the summary is then calculated. ALOHA takes these rubrics into a web-based tool and adds features that also facilitate the grading process for the TAs. The grading view of ALOHA which holds the rubric inside is shown in Figure 5.4.

Besides having an existing grading scheme to help forming the grade, the TAs are provided a feature called *semi-automatic phrasing*. This helps in generating proper, personalized written feedback which is sent for the student. While filling the grading rubric (giving grades for small sub-parts) the TA already selects suitable predefined phrases/phrase templates for each topic worthwhile mentioning and then if needed,

those are manually customized to fit the correct use case. The purpose is to avoid writing the same feedback all over again for each submission, which would be very typical when grading small student programming assignments.

Besides introducing the ALOHA tool, the publication does an initiative evaluation on the use of such tool. The tool was used in Tampere University of Technology during spring semester 2006 on four programming courses. Altogether around 20 TAs had used the tool. The general opinion towards the process and the tool was very positive but no official survey was conducted on the attitudes towards the tool.

What was done was a statistical analysis to measure the effects of the tool to objectivity. The grading distributions of the same course in two years (one with the tool, one without) were examined. For each grader, a mean value of all given grades was calculated and those were analysed by variance analysis.

In the year where the tool was not used, there was a statistically significant difference between three graders. This means that if a student was lucky enough to get a certain grader for his work, he would have statistically had a much better grade than in the case of two other graders. When the tool was taken into use, this problem did not exist any more. The use of tool seems to limit the most lenient and strictest graders.

On the basis of only one comparison between two years, the result could not have been generalized but it was a good start. The real outcome of the publication is the discussion on how such tool can be used to tackle problems related grading objectively, generating feedback and managing the whole process of manually grading hundreds of student submissions.

According to Google Scholar [85] statistics, publication (vi) has been cited 8 times during 2008 and 2014.

Publication (vii), *Analyzing the Use of a Rubric-Based Grading Tool*, is a direct continuum study for publication (vi). Where publication (vi) introduced the idea of a rubric-based grading tool and the tool ALOHA, this publication examines the situation after two full years of ALOHA being used. It provides a thorough examination on how teaching assistants use a rubric-based grading tool and discusses whether the tool was used like it was intended to be used.

For the study, all given grades and feedback mails were gathered from a large programming assignment of the a programming course in Tampere University of Technology. The course was the same which was already used in publication (vi) for studying objectivity in given grades amongst different graders.

The same objectivity study than in publication (vi) was repeated to back up the earlier findings: the grading remained objective. Besides this, new measurements related to TAs work were also carried out.

Time spent with the tool for each submission by a TA was measured. Purpose was to check did the tool "force" the TAs to spend sufficient amount of time grading each individual student submission. The only data in hand for this were the timestamps of the tool log files, so only estimates were made.

The quality of the feedback mails sent for the student were also checked. As the real "quality" would require qualitative analysis, to tell whether the feedback is written "well" or "badly", the publication presents two quantitative measures to give indisputable facts on the feedback mails. Amount of lines and percentage of predefined phrases. Amount of lines was measured simple to see whether students receive "enough" or even "much" feedback from their work[2].

Percentage of pre-defined phrases is then related to the feature *semi-automated phrasing*, to see whether the TAs are using the feature or not, and how much they are writing completely custom feedback. This is not a measurement of better or worse feedback: If the percentage of pre-defined phrases in the feedback mail is high, then TA has used the tool efficiently but customised only little. The feedback can still be of high quality, as long as the phrases are of high quality.

The percentage was measured by using a sophisticated plagiarism detection tool Nalkki [88], by comparing each feedback mail against a file with all those pre-defined phrases. In general, the TAs used the pre-defined phrases well, in average a bit over 50% of all text consisted of pre-defined phrases. However, there were huge variations between different TAs, as one TAs feedback mails had the value over 90%. This indicates that a such feature clearly requires clarification, more guidance and control and phrases that require customization.

As a general outcome, the publication concludes that even with such tool the TAs work very differently which is of course natural. It is still essentially relieving that the following conclusions can be made out from these results:

- All the graders used enough time with the tool. On the other hand, no one seemed to use too much time either.
- The TAs wrote proper feedback and efficiently used predefined phrases for facilitating their workload and to ensure feedback of good quality.
- The grading remained objective when rubrics where used.

Altogether the grading tool seemed to be used—even though not homogeneously—but at least in between the limits of what was expected. The publication also dis-

[2]roughly one could say that more is better here

cusses different approaches that could be used to improve the use of such tool as some real problems were observed. Partly based on these results, a successor for ALOHA called Rubyric has been developed in Helsinki University of Technology [10].

According to Google Scholar [85] statistics, publication (vii) has been cited 8 times during 2009 and 2014.

6. CONCLUSIONS

This chapter presents the results of the work being summarised and combined. The results are directly discussed in relation to the posed research questions in the following two subsections. After that, we combine and summarize the results and discuss the benefits, limitations and generalizability of the results.

The main research question of this work, **"How can the teaching of programming be assisted with teaching technologies?"**, is discussed through the two main focus areas: efficient integration of visualization systems and tools-assisted assessment.

6.1 Efficient Integration of Visualization Systems

Here, we will combine the conclusions from our publications to answer our first research question

RQ 1, *"How can visualization systems be effectively integrated into programming courses?"*

Publications (i), (ii), (iii) and (iv) presented concrete answers by addressing the following three sub-questions:

RQ 1.1 How can program visualization examples support also deeper learning?

Publication (i) presents various learning taxonomies in CS education context and discusses what deeper cognition actually means in this context. From the point of this thesis, it describes the practical use cases and CS-specific adaptation of Bloom's Taxonomy of Cognitive Domain. As the publication concludes, the levels of Bloom's taxonomy are well present, agreeable and field-tested also in CS education. Their order and hierarchical dependency just seems to be more complex than in the original taxonomy. This is mainly because producing and interpreting programming code are separate skills. Regardless, Bloom's taxonomy provides a commonly accepted basis on which to build other categorizations and applications on.

Based on Bloom's taxonomy, publication (ii) presented a categorization of visualization examples that are based on corresponding levels of the taxonomy. Idea is to apply the same principles already applied for Algorithm Visualizations through the Engagement Taxonomy: By increasing the level of engagement in the examples the student can progress on the levels of Bloom's taxonomy with visualization examples.

For program visualizations it was required to design completely new kinds of examples as PV was not used in such high abstraction level as algorithm visualizations but more on concrete smaller items, and only to introduce the concept, thus corresponding only the first levels of cognition. Publication (ii) presented also concrete ideas for the visualization examples to reach higher levels of cognition. Publication (iii) shows a concrete case study where new kinds of visualization examples, now reaching levels 3 and 4 in Bloom's taxonomy (Application and Analysis): *illustrative* and *problem-solving visualizations* were used successfully in a basic programming course. Implementing these examples required minor modifications to the existing PV tool [54].

Visualizations can also support deeper learning of programming from another aspect, from showcasing the whole *process* and different structures of programming as we have shown in publication (iv). The whole idea and realistic process of programming (problem-solving, debugging, algorithm design, etc.) were demonstrated to programming students in their very first lecture without telling anything else about programming. Again, to deepen also the cognitive part, engaging visualizations were successfully used.

RQ 1.2 How to integrate visualization tools into student homework assignments?

Publication (iii) demonstrated a practical case study where the new visualization categories defined in publication (ii) were taken into use. The new, cognitively-more-deeper visualization examples allowed improving consistent usage of visualizations as they could be used also when preparing for exercise sessions and for submitting homework. By having interruptive questions and fill-in-the-blanks or "fix broken code" types of assignments, the homework based on visualization tools were able to go deeper into proper homework assignments. On the other hand, this also introduced more opportunities for the students to work with the tool becoming more familiar with it and making its use more consistent during the course. Based on our results, it can be concluded that the usage of visualization examples can work well as student homework if the examples are engaging enough and provide cognitively deeper challenges.

RQ 1.3 With proper integration from both cognitive and practical perspective, are visualization systems helping to learn programming?

The empirical studies around publications (iii) and (iv) show that visualization tools help the students in learning programming, especially the ones that are not yet familiar with the topics. Publication (iii) concludes that there was a statistically significant difference of the post-test results between the students who used visualizations and the ones that did not when looking at the students who found the course topics difficult. As a side effect, the visualization group also spent more time studying the topic, so the results might be partially explained by that. Also, the experiment of starting the whole course with "visualizations first", shown in publication (iv) was successful. The students found the example informative and used the engaging example to experiment the big picture of programming themselves.

From the results of publications (iii) and (iv) we can conclude that with consistent usage and proper course integration, visualization systems help students learn programming either directly by having a positive effect on cognition or at least indirectly by providing them a motivating and engaging environment to study.

6.2 Facilitating the Assessment of Programming Assignments

Our second research question, around tools-assisted assessment, was

RQ 2, *How can the assessment process be assisted with teaching technologies?*

This question is addressed in publications (v), (vi) and (vii) by answering the following three sub-questions:

RQ 2.1 What kind of teaching technologies there are in terms of features for the computer-assisted assessment of programming assignments?

Publication (v) is a survey of Computer-Assisted Assessment tools that are being used in programming courses. The general breadth of the tools is wide with a lot of independent tools being out there, mostly being developed for individual purposes. The general direction and features within the tools are following the findings encouraged in the relevant research literature. These include for instance supporting different *resubmission policies* and possibility for additional *manual assessment* to apply semi-automatic assessment processes. The publication presents a systematic categorization of the features that exist in computer-assisted assessment tools. Besides the above-mentioned, the tools have different ways for defining tests, integrating into learning management systems and sandboxing models.

The variety of tools is partially also explained by them differentiating in certain programming paradigms or programming languages even though majority of the tools were targeted only for Java development. For CS educators the wide language support is good news because they have a good amount of options to select the tools that best fit their existing teaching goals and purposes instead of needing to design their courses "tools first".

The general trend of the tools indicates that web centrality is the dominant approach facilitating the submission practicalities. Also, the integration to existing learning management systems (like Moodle) has gotten a lot more attention making it easier for the teacher to integrate these tools into the course infrastructure.

As recommendations to tool developers to get wider adoption for their tool we recommend the following focus areas that were clearly problematic when trying to get a proper grasp of the existing tools:

- transparency and explicit, clear explanations on how the system works with emphasis on examples
- proper security model
- at least partial open-source model allowing easy extendibility and customization.

As software development processes in general are moving from traditional waterfall model towards leaner agile methods and processes, it is encouraging to see these paradigms being supported by the tools as well. For instance, Web-CAT [27] is highly emphasizing the *Test-Driven Development* paradigm by allowing the students to submit their own test cases for their application along with their code. Also, web applications are becoming more important and the educational tools will need to support these paradigms better. A brief summative list for teachers of the CAA tools covered in the publication (v) is presented in its Appendix A.

RQ 2.2 How can manual assessment be assisted with rubric-based assessment tools?

Rubrics in general provide a good tool for graders to define an objective grading scheme. Especially in the case where multiple graders are used, they provide a unified and objective scheme for everyone to use consistently. There are two possible problems with traditional, paper rubrics: Firstly, all the graders might not use them consistently possibly skewing the grading into a certain direction and hindering the overall inter-consistency between graders as shown in Publication (vi). Secondly, as students need constructive and personalized feedback from their programming

assignments, the grader needs to separately write the feedback manually, re-writing the same issues several times.

Publications (vi) and (vii) show that when rubrics are used through a tool, for instance via a separate web-based rubric-tool or as an integrated part within a CAA system, the tool subtly *forces* the grader to go through each step of the rubric carefully. Especially if the cumulative formation of the final grade is not shown while going through the rubric, the graders *personal hunch* of the final grade will not skew the results. However, we feel that the important aspect why our results were positive were because the graders found the tool to be helpful for them, making their job easier by providing features that facilitate in writing the feedback. In this case, the tool provided semi-automated phrasing which helped generating feedback based on the individual marks in the rubric.

For the teacher who is responsible for building the grading scheme, the rubric-based assessment tools certainly create extra labour. Besides defining the scheme, splitting it into small sub-parts of the rubric and inputting the rubric to the tool somehow (for instance by writing an XML file) he also needs to create all the pre-defined phrases that are to be used with semi-automated phrasing. So the teacher is not responsible only for the scheme but also for what kind of feedback should be written for each part. Depending on the level of seniority of the graders this might be a desired feature. Junior teaching assistants certainly require closer guidance in their grading from the teacher than senior colleagues. Naturally, the teacher can himself adjust the amount of effort he puts into the pre-defined phrases and how much freedom does he leave for the graders themselves. If the same course has multiple iterations, the previous schemes can be reused over time.

In the experiments, described in publications (vi) and (vii) the tool let teaching assistants create their own phrases if they found out something was missing or felt like contributing. Then, the best ones were shared to the others immediately and used as default ones for the following iteration of the course.

RQ 2.3 Are rubric-based assessment tools effective and being used as they ideally should be?

As shown in publications (vi) and (vii), when traditional paper rubrics were used in a programming course with multiple graders there was a clear statistical difference between the grades given by different graders: the grading was not consistent and objective. When looking at the results of the following two years when a rubrics-based tool was used, those problems of were gone and the grading did not have statistical differences between the graders. We can clearly conclude the result to be positive.

What was also analysed in publication (vii) was the detailed usage statistics of the tool: how did the teaching assistants use the tool in terms of time, days on the task and how semi-automated phrasing was used in writing the feedback? The results showed that there were clear differences in how the nine teaching assistants acting as graders used the tool. The overall mean times spent with the tool for each submission varied between 18 and 44 minutes. Some of the difference could be explained by the different experience levels of the graders but it also indicates that some graders first spent more time with the program code then moving on to the rubric tool while some went through the code while filling the rubric simultaneously. Regardless, the mean times all fit into the assumptions and guidance from the teacher on how much to spend time with each submission[1] and the idea was not to force the graders into one harmonized working process. Looking at the mean times for each assessment over time, it was found out that grading got slightly quicker as the graders got experienced with the tool which was an expected and hoped feature.

Looking at the quality of the feedback, purely from the perspective of quantitative measures, the length of feedback written with the tool was overwhelmingly large. Each student received multiple pages of personalized feedback from their submissions. We did not measure how the students felt about the quality of the feedback but the overall course feedback indicated that after multiple weeks of hard work towards their submission they appreciated receiving properly written feedback from the course staff.

Semi-automated phrasing also seemed to be a feature that the graders both appreciated and utilized as we hoped. When comparing the written feedbacks to all of the pre-defined phrases using a plagiarism tool (to find out correlation) all graders had used the feature extensively, but still customizing the feedback at least around 30%, average being around 50–50 between pre-defined phrases and custom feedback. Combining the length of the feedback, time used with the tool and the usage of pre-defined phrases we can conclude that a rubric-based tool with semi-automated phrasing helps the graders in writing a very good amount of personalized feedback in a very reasonable time.

In general, the teaching assistants used the tool the way they were expected but there were quite big differences between the different graders in their grading process. This is an aspect that the teacher needs to consider when introducing such a tool into use: The graders need to be properly introduced to the way the tool is supposed to be used but especially if more senior graders are used they should not lose their freedom of defining their own working strategies. Rubric-based assessment tools are an effective aid for the teaching stuff but are not meant to replace the know-how of well-trained teaching staff.

[1]...as well as what was the basis for paying the graders for each assessment.

6.3 Summary of the Results

In this thesis, we have discussed how the teaching of programming can be efficiently facilitated by different teaching technologies, focusing especially on visualization systems and tools-assisted assessment. The combined results and a conclusive model on the role of these technologies in different parts of the student's learning path within a programming course are shown in Figure 6.1, which we will now walk through.

Visualization systems can be used as an effective additional mean for learning programming. When being introduced to new topics, illustrative visualizations can be used, for instance within the lectures by the teacher presenting them. Having only simple illustrative visualizations shown occasionally to the students is not enough. Successful use of visualization systems requires proper integration to the whole learning process. Instead of just using illustrative non-engaging visualization examples the student should be engaged to interact with the examples. This has two main benefits: Through better engagement the student's cognitive development within the subject is enhanced, and there are more learning situations where visualization examples can be used. Consistent usage of visualizations within the course materials and infrastructure improves the learning process. In providing actual practical examples of cognitively-improved program visualization examples, we have used the Bloom's Taxonomy of Cognitive Development and the Engagement Taxonomy to demonstrate how visualizations can enhance the deeper levels of cognition.

As the student needs to constantly practice his programming skills, actual assessed programming assignments are a vital part of a programming course as assessment greatly guides the student focus and efforts during the course. As a lot of programming courses are organized as mass courses with hundreds of students it is impractical for the teacher to manually assess student programs. Computer-assisted assessment tools allow the teacher to provide an easily accessible environment for the students to submit their programming assignments and to receive formative (or summative) feedback from their work. There are a lot of existing CAA tools that can be used for different programming languages with different features and configuration possibilities that allow the teacher to adjust the tool to their exact course needs.

Important part of learning programming is also learning the proper process for creating software, including design, implementation, coding style, testing and debugging. As good as CAA tools are, they are not sufficient for testing everything the student needs to learn. Semi-automated assessment process uses CAA for everything that can be automatically tested but applies manual grading for the parts where human perspective is needed for assessment and for personalized, construc-

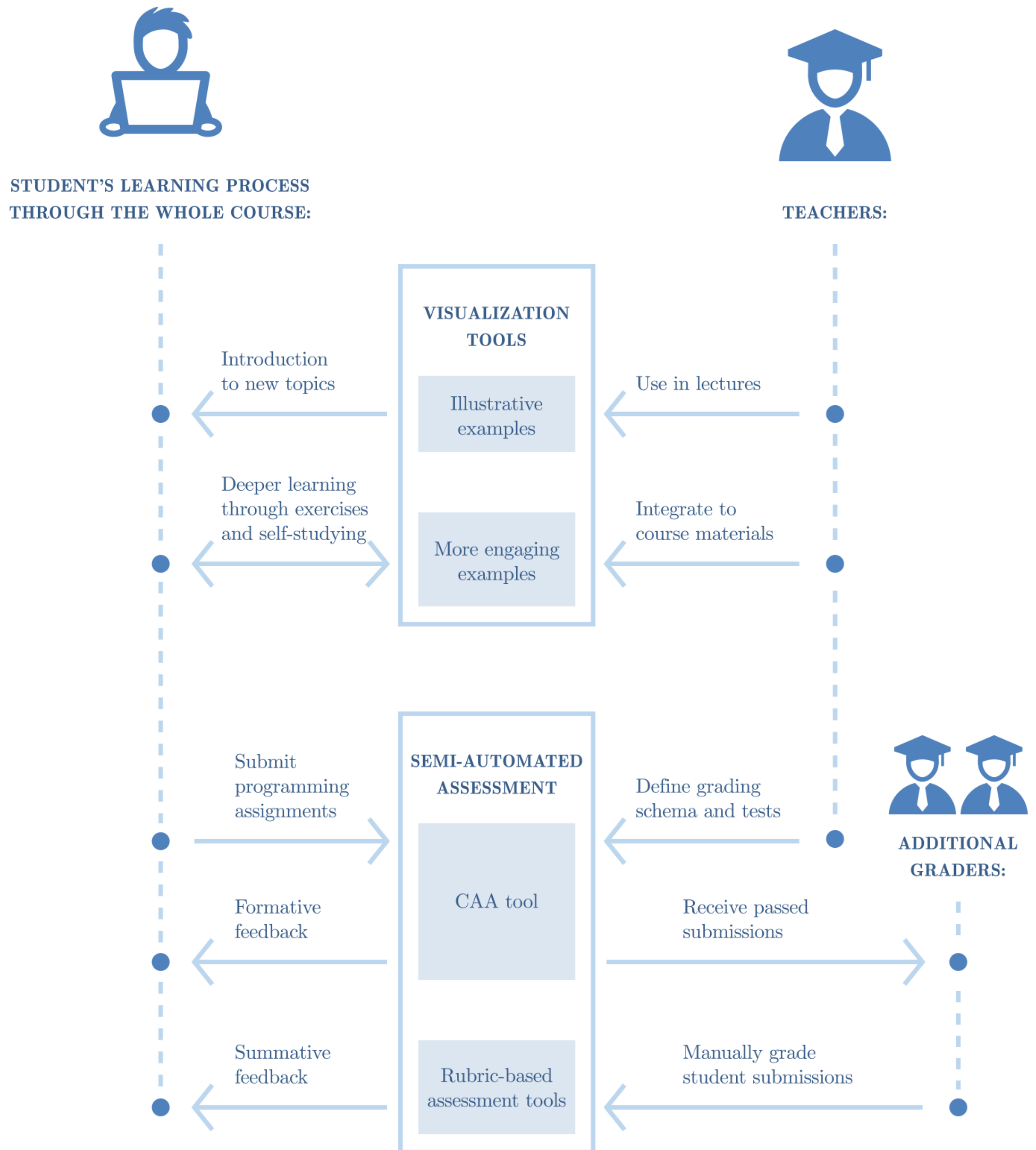


Figure 6.1: Assisting the student learning process with different teaching technologies during the whole programming course.

tive feedback. Rubric-based assessment tools are a useful aid for manual grading to provide consistency and objectivity in the grading especially when multiple graders are being used. They can also include convenience features that make the grading process more effortless and help in generating good feedback for the students.

Regardless of the teaching technologies that are selected to being used by the teacher, it is important that these tools are meant to *facilitate* and *support* the practical work of the teacher and help in the student's learning process. They are not meant to replace the skilled teachers or graders and the importance of their work in supporting the students in their process of slowly but steadily becoming expert programmers.

6.4 Generalization and Limitations of the Work

The results of the work are gathered through individual experiments but have been presented in this thesis in already rather generalized format. In this thesis, we have placed our own work within the large area of similar research thus getting the confidence to generalize our findings enough to make somewhat general recommendations for teachers. Certain limitations and self-criticism however must be explicitly mentioned here.

Teachers and researchers have different opinions on the different levels of learning taxonomies, especially when adapted into practical examples. Our adaptation of Bloom's taxonomy in to CS context and the visualization categories based on the original Bloom's taxonomy can certainly be disputed to some extend. The CS-specific adaptation of Bloom's is applicable enough to be used to explain the phenomena of earlier research results discussed in publication (i) and is supposed to work as a mean of communication, as a common vocabulary, and a tool for teachers to understand the learning paths their students take. For the new visualization categories, our intention was not to provide an ultimate hierarchical taxonomy but instead present new program visualization types that are based on the relevant levels of Bloom's taxonomy, regardless of their ultimate order.

The data we have collected and presented to support the effects of visualization usage in exercise sessions is statistically meaningful but certainly limited to the setup we used. In different courses, with different teaching staff, topics and practical implementations variance between results is expected. Thus, we feel we cannot make a generalized statement that program visualizations always improve learning when used properly over the course. Our results are however only positive. This is also the mixed opinion in the existing research: Even though sometimes results have not been clear, there are a lot of positive experiences with using visualizations, and usually the relative difference in results comes from *how* they are used, which is what we want to emphasize too.

As stated, our survey of existing CAA tools and their features in publication (v) must be treated as a continuum study to the one of Ala-Mutka from 2005 [6]. Thus, our work does not contain the tools that were already used before 2005. Also, despite our large literature review, there must be a lot of CAA tools we have not covered.

For our results on the use of rubric-based assessment tools we were able to replicate our data analysis presented in publication (vi) also in publication (vii). With both of them showing positive results, we feel confident in saying our approach did improve objectivity and inter-grader consistency in the manual assessment. Again, when taken into other courses with different teaching staff and grading schema, the results can be different so generalization is again to be treated with a disclaimer. But at least, the situation should be improved assuming the rubric-based tool is being used following the grading schema.

Finally, one cause of possible criticism towards our work is that the included publications in this thesis in general are at least couple of years old each, some even close to ten years, and the original research in them does not include the most recent years. We acknowledge that a lot has certainly happened within CSER during those years. The introduction chapters of this thesis are up-to-date and provide insight to even the latest research in the field making this work as a whole a 10 years worth of research. Also, in this case, time has served our research well. Looking at our work after a while has given the surrounding research field time to react on it and based on the citations and discussion our work has gotten, its validity within the field has been positive. The intention of our work has not been to provide an individual set of teaching technologies to recommend but to discuss the approaches in adopting any of the existing or future tools, making our results last time better.

6.5 Benefits of the Work

Besides presenting the conclusions earlier we also want to emphasize what sort of an impact do we wish to contribute with this work to different *stakeholders*, teachers, teaching technology developers, CS education researchers and of course students.

Benefits for Teachers

We shall start with teachers, as this work is mainly targeted to be a thorough, but sufficiently *bite-sized* overview for CS teachers on the offering that CS education research has provided to facilitate parts of their practical, every-day work. We see a lot of contribution to the teachers.

Firstly, we want teachers to recognize that there is a full research discipline trying to tackle the problems they face every day. Despite being a science discipline, CS

education research is also very close to practice, like this work, providing practical but research-based solutions to real problems.

Secondly, if unfamiliar to the research discipline, it is very difficult to approach the *plethora* of different tool offering, and even though something would be found, adopting the tool efficiently to the curricula requires knowing the best practices. Here is where we see our biggest contribution: The work introduces the field offering, researched-verified results and also the best practices to efficiently integrate tools teaching technologies into a programming course.

Benefits for Teaching Technology Developers

We also see contribution towards people developing these tools. Especially we want the people who consider creating their own tools to first open their eyes for the already existing offering. Both software visualizations and assessment tools have developed enormously over the past decades and the general opinion in the research is widely leaning towards collaboration and integration of the existing tools in order to create larger ecosystems instead of individual silos. Certainly, especially as the field of CS evolves, there will always be need for new things to visualize or assess but instead of making completely new tools from the scratch, one should first investigate whether an existing tool ecosystem could be extended through collaboration towards the new features. Unfortunately, this is not always possible and it is understandable why new small tools get created. Regardless, we hope to contribute at least in advising what sort of features there are, what are the results of using them and how people have already tackled certain issues the first phases of the tools might have had.

As a concrete examples, for program visualization tool developers, we are contributing validated results on creating systems and examples that engage the student by the new visualization examples that we have created based on the Bloom's taxonomy, shown in publications (ii) and (iii). For CAA tools, publication (v) provides a survey of features of such tools to take into account when developing one's own. For rubric-based assessment tools, we have provided data on how our original approached were used with recommendations to move forward. Already using these results, succeeding tools have been developed.

Benefits for CS researchers

Besides good ground work for either visualization research or around tools-assisted assessment we find some benefits especially from the CS-specific adaptation of Bloom's taxonomy, the Matrix Taxonomy, presented in publication (i). Already

now, at the time of this thesis, the article has been out for several years and has been widely cited and used as a basis for the everlasting discussion around suitability of certain learning taxonomies. As has been argued, the new CS-specific Matrix Taxonomy is certainly not fitting all possible use cases, but it does provide one fresh look into the real-life problems students face when learning to program. As such, the new taxonomy does provide a useful tool for further research—which already now, has been quite active around the topic.

Certainly, the other findings of our work provide a lot of possibilities for future work in research as well. For instance, the new visualization categories give a good basis but certainly can use of further validation, new concrete implementations and eventually incremental iterations.

Benefits for the Student

This thesis is not directly targeted for the student even though the student is in the lead role here most of the time. The tools are meant to be used by students but rarely it is the student who has a saying on the tools used for the course infrastructure. Naturally, these tools are targeted for the student to learn better, so we hope that our ultimate biggest contribution will be towards students even though our results directly are not targeted for students to apply.

Sometimes, the students become teaching assistants that are grading submissions of other students. Eventually they might end up becoming teachers and even CS education researchers. Maybe, if read by a student, this work could also contribute by planting a small seed for better educational practices. But mostly, until then, *students*, we are happy to see you have selected to take the journey towards becoming expert programmers. You are, *after all*, why CS teachers and CS education researchers do their work. So, keep on rocking! *(See Figure 6.2)*

Figure 6.2: The author of this thesis, rocking. Thank You!

BIBLIOGRAPHY

[1] Tuukka Ahoniemi, Essi Lahtinen, and Teemu Erkkola. Fighting the student dropout rate with an incremental programming assignment. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, Koli Calling '07, pages 163–166, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[2] Tuukka Ahoniemi, Essi Lahtinen, and Keeko Valaskala. Why should we bore students when teaching CS? In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research*, November 2007.

[3] Tuukka Ahoniemi and Tommi Reinikainen. ALOHA - a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, Baltic Sea '06, pages 139–140, New York, NY, USA, 2006. ACM.

[4] Kirsti Ala-Mutka. *Automatic Assessment Tools in Learning and Teaching Programming*. PhD thesis, Tampere University of Technology, 2005.

[5] Kirsti Ala-Mutka and Hannu-Matti Järvinen. Assessment process for programming assignments. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies, ICALT 2004, 30 August - 1 September 2004, Joensuu, Finland*, page 0, 2004.

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

[7] 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 Revision of Bloom's Taxonomy of Educational Objectives, Abridged Edition*. Allyn & Bacon, 2000.

[8] A. Anglin, K. Anglin, P. L. Schumann, and J. A. Kaliski. Improving the efficiency and effectiveness of grading through the use of computer-assisted grading rubrics. *Decision Sciences Journal of Innovative Education*, 6:51–73, 03/2008 2008.

[9] Doug Atkinson and Siew Leng Lim. Improving assessment processes in higher education: Student and teacher perceptions of the effectiveness of a rubric embedded in a lms. *ICALT '13: Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, 2013.

[10] Tapio Auvinen, Ville Karavirta, and Tuukka Ahoniemi. Rubyric: an online assessment tool for effortless authoring of personalized feedback. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 377–377, New York, NY, USA, 2009. ACM.

[11] Ronald Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. MIT Press, Cambridge, MA, USA, 1997.

[12] Katrin Becker. Grading programming assignments using rubrics. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, ITiCSE '03, pages 253–253, New York, NY, USA, 2003. ACM.

[13] Roman Bednarik. *Methods to analyze visual attention strategies: Applications in the studies of programming.* PhD thesis, University of Joensuu, Department of Computer Science and Statistics, 2007.

[14] Roman Bednarik, Niko Myller, Erkki Sutinen, and Markku Tukiainen. Analyzing Individual Differences in Program Comprehension. *Technology, Instruction, Cognition and Learning*, 3(3):205–232, 2006.

[15] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, 22(5):375 – 384, 2011.

[16] S D Benford, E K Burke, E Foxley, and C A Higgins. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd Annual on Southeast Regional Conference*, ACM-SE 33, pages 176–182, New York, NY, USA, 1995. ACM.

[17] John Biggs and Catherine Tang. *Teaching for Quality Learning at University, fourth edition.* Open Univ Press, 2011.

[18] B.S. Bloom, M.D. Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl. *Taxonomy of Educational Objectives: Handbook 1 Cognitive Domain.* Longmans, Green and Co Ltd, London, 1956.

[19] Marc H. Brown. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings of IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[20] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, January 1984.

[21] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. How shall we assess this? In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '03, pages 107–123, New York, NY, USA, 2003. ACM.

[22] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.

[23] Gil Ebel and Mordechai Ben-Ari. Affective effects of program visualization. In *Proceedings of the second international workshop on Computing education research*, ICER '06, pages 1–5, New York, NY, USA, 2006. ACM.

[24] Bob Edmison, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. Using a rubric-based assessment system to improve feedback and student performance in course-management systems. In *Proceedings of 2010 ASEE Southeast Section Conference*, 2010.

[25] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3), September 2003.

[26] 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(4):167–182, November 2008.

[27] Stephen H. Edwards and Manuel A. Perez-Quinones. Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ITiCSE '08, pages 328–328, New York, NY, USA, 2008. ACM.

[28] Sally Fincher and Marian Petre. *Computer Science Education Research*. Taylor & Francis, 2004.

[29] G. Gibbs. Using assessment strategically to change the way students learn. In *Assessment Matters in Higher Education: Choosing and Using Diverse Approaches*. 1999.

[30] S. Habeshaw, G. Gibbs, and T. Habeshaw. *53 Problems with Large Classes: Making the Best of a Bad Job*. Interesting ways to teach. Technical and Educational Services, 1992.

[31] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The coursemarker cba system: Improvements over ceilidh. *Education and Information Technologies*, 8(3):287–304, September 2003.

[32] Christopher D. Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach. *Comput. Educ.*, 39(3):237–260, November 2002.

[33] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259 – 290, 2002.

[34] Essi Isohanni and Hannu-Matti Järvinen. Are visualization tools used in programming education?: By whom, how, why, and why not? In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 35–40, New York, NY, USA, 2014. ACM.

[35] Essi Isohanni and Maria Knobelsdorf. Behind the curtain: Students' use of vip after class. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pages 87–96, New York, NY, USA, 2010. ACM.

[36] David Jackson. A semi-automated approach to online assessment. *SIGCSE Bull.*, 32(3):164–167, July 2000.

[37] David Jackson and Michelle Usher. Grading student programs using assyst. *SIGCSE Bull.*, 29(1):335–339, March 1997.

[38] Colin G. Johnson and Ursula Fuller. Is bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, Baltic Sea '06, pages 120–123, New York, NY, USA, 2006. ACM.

[39] Mike Joy, Nathan Griffiths, and Russell Boyatt. The boss online submission and assessment system. *J. Educ. Resour. Comput.*, 5(3), September 2005.

[40] Harri Järvi. Visualisointiympäristö ohjelmoinnin perusopetukseen. Master's thesis, Tampere University of Technology, 2010.

[41] Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, and Tapio Salakoski. Automatic assessment of program visualization exercises. In *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, pages 101–104, New York, NY, USA, 2008. ACM.

[42] Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, and Tapio Salakoski. Effects of course-long use of a program visualization tool. In *Proceedings of the Twelfth Australasian Conference on Computing Education - Volume 103*, ACE '10, pages 97–106, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[43] Ville Karavirta. *Facilitating Algorithm Visualization Creation and Adoption in Education*. PhD thesis, Teknillinen korkeakoulu, 2009.

[44] Ville Karavirta and Ari Korhonen. Automatic tutoring question generation during algorithm simulation. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, Baltic Sea '06, pages 95–100, New York, NY, USA, 2006. ACM.

[45] Ville Karavirta, Ari Korhonen, Lauri Malmi, and K. Stalnacke. MatrixPro - a tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33, University of Warwick, UK, 07/2004 2004.

[46] Colleen Kehoe, John Stasko, and Ashley Taylor. Rethinking the evaluation of algorithm animations as learning aids An observational study. *International Journal of Human-Computer Studies*, 54:265–284, 1999.

[47] Michael Kolling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[48] Ari Korhonen, Mikko-Jussi Laakso, and Niko Myller. How does algorithm visualization affect collaboration? Video analysis of engagement and discussions. In *WEBIST 2009 - 5th International Conference on Web Information Systems and Technologies*, 2009.

[49] Ari Korhonen and Lauri Malmi. Matrix - concept animation and algorithm simulation system. In Levialdi S. Panizzi E. De Marsico, M., editor, *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114. ACM, Trento, Italy, 2002.

[50] Mikko-Jussi Laakso. *Promoting Programming Learning–Engagement, Automatic Assessment with Immediate Feedback in Visualizations*. PhD thesis, University of Turku, Turku Center for Computer Science (TUCS), 2010.

[51] Essi Lahtinen. Integrating the use of visualizations to teaching programming. In Hannu-Matti Järvinen and Kirsti Ala-Mutka, editors, *MMT 2006 Methods, Materials and Tools for Programming Education Conference Proceedings*, 2006.

[52] Essi Lahtinen. A categorization of novice programmers: A cluster analysis study. In *Proceedings of the 19th annual Workshop of the Psychology of Programming Interest Group*, 2007.

[53] Essi Lahtinen. Students' individual differences in using visualizations: Prospects of future research on program visualizations. In *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, pages 92–95, New York, NY, USA, 2008. ACM.

[54] Essi Lahtinen and Tuukka Ahoniemi. Annotations for defining interactive instructions to interpreter based program visualization tools. *Electron. Notes Theor. Comput. Sci.*, 178:121–128, July 2007.

[55] Essi Lahtinen, Tuukka Ahoniemi, and Anniina Salo. Effectiveness of integrating program visualizations to a programming course. In Raymond Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 195–198, Koli National Park, Finland, 2007. ACS.

[56] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3):14–18, June 2005.

[57] Essi Lahtinen, Hannu-Matti Järvinen, and Suvi Melakoski-Vistbacka. Targeting program visualizations. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '07, pages 256–260, New York, NY, USA, 2007. ACM.

[58] S.-P. Lahtinen, E. Sutinen, and J. Tarhio. Automated animation of algorithms with Eliot. *Journal of Visual Languages & Computing*, 9(3):337 – 349, 1998.

[59] Ronit Ben-Bassat Levy and Mordechai Ben-Ari. We work so hard and they don't use it: acceptance of software tools by teachers. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '07, pages 246–250, New York, NY, USA, 2007. ACM.

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

[61] Raymond Lister and John Leaney. First year programming: Let all the flowers bloom. In *Proceedings of the Fifth Australasian Conference on Computing*

Education - Volume 20, ACE '03, pages 221–230, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[62] Raymond Lister and John Leaney. Introductory programming, criterion-referencing, and bloom. *SIGCSE Bull.*, 35(1):143–147, January 2003.

[63] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bull.*, 38(3):118–122, June 2006.

[64] Harri Luoma. Tulkin toteutus ohjelmoinnin perusopetuksen tarpeisiin. Master's thesis, Tampereen teknillinen yliopisto, 2007.

[65] Harri Luoma, Essi Lahtinen, and Hannu-Matti Järvinen. CLIP, a Command Line InterPreter for a subset of C++. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research*, November 2007.

[66] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppälä, and Panu Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[67] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33(4):125–180, December 2001.

[68] Orni Meerbaum-Salant, Michal Armoni, and Mordechai (Moti) Ben-Ari. Learning computer science concepts with scratch. In *Proceedings of the Sixth International Workshop on Computing Education Research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM.

[69] Andrés Moreno, Niko Myller, Mordechai Ben-Ari, and Erkki Sutinen. Program animation in jeliot 3. *SIGCSE Bull.*, 36(3):265–265, June 2004.

[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, March 2009.

[71] Niko Myller, Mikko Laakso, and Ari Korhonen. Analyzing engagement taxonomy in collaborative algorithm visualization. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '07, pages 251–255, New York, NY, USA, 2007. ACM.

[72] Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVE-an environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, SIGCSE '00, pages 109–113, New York, NY, USA, 2000. ACM.

[73] Thomas L. Naps, Guido Rössling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '02, pages 131–152, New York, NY, USA, 2002. ACM.

[74] Seppo Nevalainen and Jorma Sajaniemi. An experiment on short-term effects of animated versus static visualization of operations on program perception. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 7–16, New York, NY, USA, 2006. ACM.

[75] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.

[76] Arnold Pears, Stephen Seidman, Crystal Eney, Päivi Kinnunen, and Lauri Malmi. Constructing a core literature for computing education research. *SIGCSE Bull.*, 37:152–161, December 2005.

[77] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39:204–223, December 2007.

[78] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

[79] B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of sofware visualization. 4(3):211 – 266, 1993.

[80] T. Rajala, E. Kaila, J. Holvitie, R. Haavisto, M. Laakso, and T. Salakoski. Comparing the collaborative and independent viewing of program visualizations. In *Frontiers in Education Conference (FIE), 2011*, pages F3G–1–F3G–7, Oct 2011.

[81] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. VILLE: A language-independent program visualization tool. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*,

Koli Calling '07, pages 151–159, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[82] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[83] Guido Rössling and Bernd Freisleben. AnimalScript: An extensible scripting language for algorithm animation. *SIGCSE Bull.*, 33(1):70–74, February 2001.

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

[85] Google Scholar. http://scholar.google.com, December 2014.

[86] Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. Algorithm visualization: A report on the state of the field. *SIGCSE Bull.*, 39(1):150–154, March 2007.

[87] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. Algorithm visualization: The state of the field. *Trans. Comput. Educ.*, 10(3):9:1–9:22, August 2010.

[88] Petri Sirkkala and Sami Puonti. Nalkki-project - tool for plagiarism detection using the web. In Raymond Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 229–230, Koli National Park, Finland, 2007. ACS.

[89] Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. PhD thesis, Aalto University, 2012.

[90] John Stasko. Smooth, continuous animation for portraying algorithms and processes. MIT Press, Cambridge, MA, USA, 1997.

[91] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning?: An empirical study and analysis. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 61–66, New York, NY, USA, 1993. ACM.

[92] John T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[93] John T. Stasko, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization*. MIT Press, Cambridge, MA, USA, 1997.

[94] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*, ITiCSE -WGR '13, pages 15–32, New York, NY, USA, 2013. ACM.

[95] Juha-Matti Vanhatupa, Arto Salminen, and Hannu-Matti Järvinen. Organizing and evaluating course on embedded programming. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 112–117, New York, NY, USA, 2010. ACM.

[96] Antti Virtanen. Visuaalinen tulkki ohjelmoinnin opetukseen. Master's thesis, Tampere University of Technology, 2005.

[97] Antti T. Virtanen, Essi Lahtinen, and Hannu-Matti Järvinen. VIP, a Visual Interpreter for Learning Introductory Programming with C++. In Tapio Salakoski, Tomi Mäntylä, and Mikko Laakso, editors, *Proceedings of Koli Calling 2005*, 2006.

[98] Leon E. Winslow. Programming pedagogy–a psychological overview. *SIGCSE Bull.*, 28(3):17–22, September 1996.

7. PUBLICATIONS REPRINTED

Publication (i)

Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T., McGee Thompson, D., Riedesel, C., Thompson, E. Developing a Computer Science-Specific Learning Taxonomy *In: Working group reports on ITiCSE on Innovation and technology in computer science education*, June 2007, Dundee, Scotland

Developing a Computer Science-specific Learning Taxonomy

Ursula Fuller
Computing Laboratory
University of Kent
Canterbury CT2 7NF
United Kingdom
U.D.Fuller@kent.ac.uk

Colin G. Johnson
Computing Laboratory
University of Kent
Canterbury CT2 7NF
United Kingdom
C.G.Johnson@kent.ac.uk

Tuukka Ahoniemi
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
tuukka.ahoniemi@tut.fi

Diana Cukierman
School of Computing Science
Simon Fraser University
Burnaby, British Columbia
Canada
diana@cs.sfu.ca

Isidoro Hernán-Losada
Lenguajes y Sistemas Informáticos
Universidad Rey Juan Carlos
Madrid
Spain
Isidoro.hernan@urjc.es

Jana Jackova
Faculty of Management Science
and Informatics
University of Zilina /Slovak
University of Technology
Zilina, Slovak Republic
Jana.Jackova@fri.uniza.sk

Essi Lahtinen
Institute of Software Systems
Tampere University of Technology
Tampere
Finland
essi.lahtinen@tut.fi

Tracy L. Lewis
Information Technology
Radford University
Radford, VA 24142
USA
Tlewis32@radford.edu

Donna McGee Thompson
Student Learning Commons
Simon Fraser University
Burnaby, British Columbia
Canada
dmcthomp@sfu.ca

Charles Riedesel
Computer Science & Engineering
University of Nebraska Lincoln
259 Avery Hall
Lincoln, Nebraska 68588-0115
USA
riedesel@cse.unl.edu

Errol Thompson
Massey University
Wellington
New Zealand
kiwiet@computer.org

ABSTRACT
Bloom's taxonomy of the cognitive domain and the SOLO taxonomy are being increasingly widely used in the design and assessment of courses, but there are some drawbacks to their use in computer science. This paper reviews the literature on educational taxonomies and their use in computer science education, identifies some of the problems that arise, proposes a new taxonomy and discusses how this can be used in application-oriented courses such as programming.

Keywords
Computer science education, taxonomies of learning, curricula, assessment, credit transfer, benchmarking

Categories and Subject Descriptors
K.3.2 Computer and Information Science Education

General Terms
None

1. INTRODUCTION

1.1 Motivation
Educational taxonomies are a useful tool in developing learning objectives and assessing student attainment. They can also be deployed in educational research, for example to classify test items and investigate the range of learning these are measuring. The well-known educational taxonomies are generic and rely on the assumption that the hierarchy of learning outcomes is the

same in all subjects, from art history to zoology. However, taxonomies are not simple to use and researchers find it hard to reach agreement on the classification of items, which limits their benefits to instructors [27]. This paper reports the work of an ITiCSE Working Group investigating the hypothesis that the hierarchy of learning outcomes in computer science is not well captured by existing generic taxonomies and that computer science education would be better served by the development of a computer science-specific taxonomy.

1.2 What is an educational taxonomy?

A taxonomy is a classification system that is ordered in some way. Linnaeus's taxonomy arranged living organisms into a tree-structured hierarchy. This gave biologists a tool to help them understand the relationship between members of the plant and animal kingdoms and to communicate accurately about them [7]. Taxonomies of educational objectives can similarly be used to provide a shared language for describing learning outcomes and performance in assessments. Unlike the biological taxonomy, educational taxonomies are not usually tree-structured. To a greater or lesser extent they divide educational objectives into three domains, *cognitive, affective* and *psychomotor.* Some, such as Bloom's taxonomy, treat each of these as a one-dimensional continuum [7], others, like the revised Bloom's taxonomy, describe the cognitive domain using a matrix [3]. Yet others, like the SOLO taxonomy, use a set of categories that describe a mixture of quantitative and qualitative differences between the performance of students [5] and there are also taxonomies that claim they can be applied equally to all three domains.

1.3 What taxonomies are used for

Learning taxonomies describe and categorize the stages in cognitive, affective and other dimensions that an individual may be at as part of a learning process. Paraphrasing Biggs [6], we can say that they help with "understanding about understanding" and "communicating about understanding". Thus learning taxonomies can be seen as a *language* which can be used in a variety of educational contexts.

Learning taxonomies can be used to define the curriculum objectives of a course, so that it is not only described on the basis of the topics to be covered, but also in terms of the desired level of understanding for each topic [48]. Computing programs accredited by ABET have to be specified in terms of measurable objectives, including expected outcomes for graduates [14]. More generally, the use of learning outcomes is mandated in the countries of the European Higher Education Area [1,8,68] and is increasingly prevalent in the US and elsewhere [15].

Learning taxonomies are widely used to describe the learning stages at which a learner is operating for a certain topic. For example, a student may be capable of reciting by heart what recursion is but not capable of implementing a recursive algorithm. An instructor may aim to have his or her students learn a topic at a certain level in a taxonomy (e.g. students may be expected to be able to comprehend the concept of recursion without necessarily applying it). Once this has been done, the instructor can assess students at the chosen level through a suitable choice of questions or examples [39]. This approach is encouraged by teacher-trainers [26]. Furthermore, the students' answers can be analyzed as belonging to one level or another; such answers can help the instructor revise his or her teaching techniques to better guide students to accomplish a certain learning stage.

Learning taxonomies have been used in many other contexts, such as introducing students to a learning taxonomy to raise their awareness and improve their level of understanding and their studying techniques [16,71]. They are also used to structure exercises in computer-based and computer-assisted instruction [21,36].

1.4 Weaknesses of taxonomies from a CS standpoint

Learning taxonomies, particularly Bloom's taxonomy of the cognitive domain, have had a considerable impact on curriculum and assessment design in the last fifty years. However, this does not mean that their use is unproblematic. The classification of a specific learning outcome or test item depends on its context. A task that challenges the analysis and synthesis skills of a beginner becomes routine application of knowledge for a more advanced learner. Similarly, a student who has been taught how to solve a problem that is extremely similar to the test item will demonstrate skills lower in the taxonomic order than one who is solving it from first principles. This is a generic problem but computer science-specific difficulties also manifest themselves.

Johnson and Fuller [27] found that colleagues disagreed about the relative difficulty of cognitive tasks in computer science. A significant proportion felt that it is easier to apply knowledge to solve simple problems than to describe this knowledge. They also found that computer science instructors did not find the terms *synthesis* and *evaluation* useful in describing learning outcomes and assessment tasks for programming courses, especially at the introductory level, instead seeing the application of knowledge as the highest skill that they should be developing. Close questioning revealed that application, as used by these colleagues, did in fact subsume analysis, synthesis and evaluation, leading Johnson and Fuller to propose a revised taxonomy with *higher application* as the highest level.

Lahtinen's recent work [37] shows that the ordering of cognitive tasks in Bloom's taxonomy is a very poor fit for the learning trajectories of some students tackling programming for the first time. In addition, the use of taxonomies is concentrated on the cognitive domain, even though learning in the affective domain is also essential for the formation of computer science practitioners. These problems led the working group to investigate whether a subject specific taxonomy would be of more use to computer science instructors than the existing generic ones.

1.5 Methodology

In order to investigate this hypothesis, our working group has reviewed a number of taxonomies described in the educational literature, together with the range of uses to which they are put. We have also reviewed studies in the computer science education research literature that use one or more taxonomies as an analytic tool. In addition we have looked at the practice of assessment in computer science both for novice programming and in two other typical subject areas, drawing on the experience of members of the working group and their colleagues. We have used this evidence to propose a new, computer science-specific taxonomy and to make recommendations about how it might be used. We concentrated on the cognitive domain because that is

the area in which there is existing research on the use of taxonomies in computer science.

2. REVIEW OF EXISTING TAXONOMIES

Educational researchers have developed a range of taxonomies, developmental stages and instructional design strategies aimed at helping educators develop learning outcomes, educational resources, and assessments. These taxonomies have been based on a range of educational theories and research. Readers interested in the theoretical foundations for the taxonomies reviewed by the working group should direct their attention to the referenced papers.

2.1 Cognitive domain

2.1.1 Bloom and revision

Of these taxonomies, the most widely cited in the literature reviewed by the working group is the original Bloom's taxonomy [7]. Bloom's taxonomy has six categories, where each category builds on the lower ones:

	1. Knowledge
Abilities and skills	2. Comprehension
	3. Application
	4. Analysis
	5. Synthesis
	6. Evaluation

Bloom's taxonomy has since been revised by Anderson et al [3]. The authors changed the nouns listed in the Bloom's model into verbs, to correspond with the ways learning objectives are typically described.

Categories	Cognitive processes
1. Remember	Recognizing, Recalling
2. Understand	Interpreting, Exemplifying, Classifying, Summarizing, Inferring, Comparing, Explaining
3. Apply	Executing, Implementing
4. Analyze	Differentiating, Organizing, Attributing
5. Evaluate	Checking, Critiquing
6. Create	Generating, Planning, Producing

Revised Bloom's taxonomy [3]

These taxonomies do not define a sequence of instruction but define levels of performance that might be expected for any given content element. A learner performing at a higher level is expected to be able to perform at the lower levels in the cognitive hierarchy. This could be interpreted as implying a sequential learning process. However, the taxonomy doesn't rule out the use of an iterative approach to learning the content.

The authors of the revised taxonomy acknowledge that there is a possible overlap in terms of the cognitive complexity among the higher level categories of the hierarchy. However, the midpoint of each of the higher level categories is seen as being more complex than the lower category [32,67]. For example, the cognitive process of Explaining in the Understand category may require a higher cognitive load than Executing in the Apply category in some contexts.

A key difference between the revised taxonomy and the original taxonomy is that the type of knowledge elements is also defined: A. Factual knowledge, B. Conceptual knowledge, C. Procedural knowledge, D. Metacognitive knowledge. This provides a matrix into which learning objectives are mapped.

Cognitive process dimension	Separate Knowledge dimension: A Factual	B Conceptual	C Procedural	D Metacognitive
1. Remember				
2. Understand				
3. Apply				
4. Analyze				
5. Evaluate				
6. Create				

Revised Bloom's Taxonomy [67]

2.1.2 Niemierko , Tollingerova, Bespalko

Other taxonomies of learning objectives have extended Bloom's taxonomy.

Niemierko and others claim that the three highest Bloom categories (higher thinking processes) cannot be ordered hierarchically in science subjects [50]. This has been used for the development of curricula in e.g. Slovakia, the Czech Republic and Poland [35]. He developed the "ABC" taxonomy of learning objectives [50] that are organized in two dimensions:

Levels	Categories of learning objectives
I. Knowledge	A. Remembering of knowledge B. Understanding of knowledge
II. Abilities and skills	C. Application of knowledge in typical problem situations D. Application of knowledge in unfamiliar problem situations

Niemierko's "ABC" taxonomy of learning objectives [50]

Applications in category D include the analysis, synthesis, and evaluation categories of Bloom's taxonomy.

Tollingerova's taxonomy [35] has five, hierarchically-ordered operation categories : 1. memory reproduction of knowledge, 2. easy thought operations with knowledge, 3. difficult thought operations with knowledge, 4. communication of knowledge, 5. creative thinking

According to Bespalko learning objectives can be expressed in two stages of abstraction and four activity levels [35, 50]:

I. Reproductive activities: 1. recognition (identification), 2. reproduction

II. Productive activities: 3. application, 4. creativity (transformation)

2.1.3 Critical thinking

Some researchers see Bloom's taxonomy as not giving enough emphasis to aspects of critical thinking. Critical thinking goes beyond the cognitive categories of the original Bloom's taxonomy to incorporate attributes of reflective judgment with respect to the value of what is being learned and to make

judgments on the reliability and authority of the associated knowledge.

The reflective judgement taxonomy by King and Kitchener has a total of seven stages that fall into three groups indicative of *pre-reflective* thought (Stages 1-3), *quasireflective* thought (Stages 4 and 5), and *reflective* thought (Stages 6 and 7) [29].

Facione's critical thinking taxonomy is more closely aligned with Bloom's taxonomy. It lists six critical thinking skills with appropriate sub-skills. To become a good critical thinker exhibiting self regulation, the person must engage in interpretation, analysis, evaluation, inference, explanation, and meta-cognitive self-regulation [18]. The revised Bloom's taxonomy [3] endeavours to capture some of these skills through the use of the knowledge dimensions and the inclusion of the meta-cognitive knowledge.

2.2 Unified domain taxonomy

There have been a number of attempts to produce a taxonomy that covers the cognitive (C), affective (A) and psychomotor (P) domains. The work of De Block [35] is an example of this approach:

1 Knowledge
  C: *Repeat, define, show, name, etc.*
  A: *Listen to opinion of others, accept notes, realize, etc.*
  P: *Show, imitate, understand sound, smell, taste, etc.*
2 Understanding
  C: *Describe, characterize, say in own words, explain, compare, etc.*
  A: *Accept opinions of others, answer questions, react to rules correctly, ask relevant questions, participate, etc.*
  P: *Demonstrate a principle, put together and disassemble something that is known, etc.*
3 Application
  C: *Solve, calculate, number, translate, illustrate, analyze, make, etc.*
  A: *React to rules automatically, accept norms and values, cooperate in a group, apply norms and rules, etc.*
  P: *Make, produce, try, repair, adapt, cook, cut, put together and disassemble something that is new, etc.*
4 Integration
  C: *Design, create, summarize, judge, decide, plan, etc.*
  A: *React to rules spontaneously, apply norms spontaneously and behave under rules, initiate cooperation, find satisfaction in behavior and work under society's rules, etc.*
  P: *Perform an activity fluently, without hesitation, without mistakes, automatically; work precisely, quickly, etc.*

Niemierko [50] describes the possibility of synthesizing an overall educational taxonomy.

2.3 Structure of the Observed Learning Outcome (SOLO)

The SOLO taxonomy makes no reference to cognitive characteristics of the learner's performance or to the affective dimension. It focuses on the content of the learner's response to what is being assessed. It endeavours to identify the nature of that content and the structural relationships within that content. The content could be designed to assess knowledge, cognitive skills, or underlying values. The taxonomy can be used to establish the relationships expected between these different types of content. It is left up to the assessor or course designer to define the type of content expected.

The SOLO levels are:

§ Prestructural – not related to topic – disjoint – missed the point
§ Unistructural – simple meaning, naming, focussing on one issue in a complex case
§ Multistructural – 'shopping list' – disorganised collection of items
§ Relational – understanding, using a concept that integrates a collection of data, understanding how to apply the concept to a familiar data set or to a problem
§ Extended abstract – relating to existing principle, so that unseen problems can be handled, going beyond existing principles [5,6].

In defining these categories, Biggs and Collis [5] use three crucial characteristics. These are:

1. capacity – how many things are handled in the content – "a quantitative increase in what is grasped" [6]
2. relating operation – the way in which the content is related to the intended purpose – the integration of the components within the content
3. consistency and closure – the drawing of conclusions or bringing to closure that is consistent

Using SOLO in assessment can provide a mechanism for holistic marking [69,70]. However, Biggs [6] provides examples of assessment strategies that use items targeted at specific SOLO levels as well as more holistic strategies.

The lower levels of the SOLO taxonomy (unistructural and multistructural) can be used to focus on individual items or attributes of what is being assessed. The higher levels with their emphasis on integration and extension of principles require a broader range of content or attributes to be examined.

The SOLO taxonomy makes no attempt to infer a cognitive processing level although it might be argued that to perform at a relational level or an extended abstract level involves greater cognitive processing than that required for unistructural or multistructural since the learners not only have to be able to recall items, they have to show the relationship among items (relational) and draw conclusions (extended abstract).

2.4 Instructional Design

Instructional designers use taxonomy concepts to guide course creation. Merrill proposed the Component Display Theory (CDT) for instructional design [46,47]. It classifies learning along two dimensions: content (facts, concepts, procedures, and principles) and performance (remembering, using, and generalizing). A complete lesson would consist of an objective followed by some combination of rules, examples, recall, practice, feedback, helps and mnemonics appropriate to the subject matter and learning task

Level of performance	Fact	Concept	Procedur e	Principle
Find				
Use				
Remember				

Type of content

The types of content are very similar to the knowledge dimensions of the revised cognitive Bloom's taxonomy [3]. The Use performance level focuses on an ability to use an existing framework to process input. The Find performance level focuses on the ability to create a new framework through the adaptation of existing rules. This has similarities to the Apply and Create categories of the revised cognitive taxonomy.

2.5 Discussion of existing taxonomies

By far the most widely used of the taxonomies reviewed above is the original work by Bloom et al. Its strengths are that it is based on extensive analysis of test items, its simplicity, and its identification of distinct, recognizable aspects of the cognitive domain. Instructors have taken it to mean that they can assess comprehension, application, analysis, synthesis and evaluation and that this hierarchy maps onto a grading scheme. The weaknesses of the original Bloom's taxonomy is that the categories have not always proved easy to apply, that there is significant overlap between the categories and debate about the order in the hierarchy of analysis, synthesis and evaluation. In addition, its simplicity means that each category combines different types of cognitive activity.

There are many variants of the original Bloom. There is evidence that the revised category names used by Anderson et al have been adopted by instructors but it is not clear that the added complexity of distinguishing aspects of the cognitive domain such as procedural and metacognitive knowledge outweighs the simplicity of the original scheme. Facione's work is similar in its approach to improving on Bloom.

The work of Niemierko, Tollingerova and Bespalko has strong similarities to Bloom but produces two separate dimensions related to knowing and applying. This addresses the difficulty of regarding Bloom's categories as a single hierarchy but does not map so nicely onto a six or seven point scale. Component display theory identifies essentially the same dimensions but is specialized for use in computer-based instruction.

SOLO is very different to the other taxonomies reviewed above because it deals with the content of the learner's response to what is being assessed. Its holistic approach means that it can be used to assess performance in the affective and psychomotor, as well as cognitive, domains. By comparison with Bloom, it may be regarded as giving less guidance to instructors because is does not map onto categories of cognitive performance that can be singled out for assessment. Its strength is in encouraging a holistic approach that supports deep learning, its weakness that there is not yet much reported experience of using it for assessment in a range of subjects.

3. THE USE OF TAXONOMIES IN COMPUTER SCIENCE EDUCATION LITERATURE

3.1 Existing Literature on Taxonomies for Computer Science

A number of papers have explored how various generic taxonomies can be applied to computing topics. In particular, there are three ways in which such taxonomies have been applied: to the design of courses at various levels of granularity in time, the design of teaching, learning and assessment materials, and, finally, the analysis of student responses to exercises. In this section, we review work on these topics.

3.1.1 Design of Courses

Some authors propose using these taxonomies for the design or evaluation of courses. Indeed, the notation of educational objectives was the original purpose of Bloom's taxonomy. This can be at a number of different granularities: it could be used for describing student progress through a single topic, through a course, or through a whole degree programme.

Howard et al. [23] propose to clearly identify goals for every lesson, and to assign them to a given level of the taxonomy. Most lessons have a number of knowledge goals, but achieving other levels varies during the course. Plotting the highest level of each level in a graph shows the evolution of the course according to knowledge depth. Scott [65] states that assessment should measure the level achieved by each student, and the grade should depend on his/her achievement. In particular, he notices that his teaching has been covering levels 3 (application) and 6 (evaluation). Buck and Stucki [11] outline an inside/out pedagogical approach based on Bloom's taxonomy for cognitive development. This framework allows students to comprehend the basic concepts before they are asked to apply them.

Doran and Langan [17] report on a project that implemented a cognitive-based approach (using Bloom's taxonomy) to the first two years of a computing degree, using strategic sequencing (spiral) and associated mastery levels of key topics. The project also investigated the use of structured closed labs, with frequent feedback and early use of teams. They used course micro-objectives mapped to specific levels in Bloom's taxonomy. Machanick [43] describe his experience of applying Bloom's Taxonomy in design three different courses.

Some applications have applied the taxonomy across a programme of study for a degree. For example, Sanders & Mueller [64] discuss the redesign of the curriculum at his university to bring material that is concerned mainly with lower Bloom levels to the early years of a degree programme, and vice versa. In other areas, Bloom's taxonomy has also been used to redesign whole curricula. In particular, Reynolds & Fox [62] extend a curriculum in Information Technology based on the ACM Curriculum'91 to include new knowledge units and describe they fit it in Bloom taxonomy levels. In the same area, Azuma et al. [4] extend this taxonomy in order to apply it to Software Engineering. Manaris & McCauley [44] presented one possible implementation of the HCI curricular guidelines

included in CC'01. This implementation employs Bloom's taxonomy to identify levels of student competence for each of the learning objectives.

Oliver et al. [51] discuss the idea of a *Bloom Rating* for courses of study. The course assessments are analysed by instructors and the level in Bloom's taxonomy that the assessment is designed to engage the students at. These are then averaged for all of the assessments on the course, and this is termed the Bloom rating. This is then applied to looking at how courses develop in the cognitive demands that they make on the students over the three years of their degree programme. They note that some modules early in the degree programme have a high rating, and some towards the end have a low rating.

This paper makes a number of assumptions about the use of Bloom's taxonomy. Firstly, that the course should develop students' cognitive skills over the (three) years of the course, engaging students at a low cognitive level at the beginning of the degree and working towards the higher levels towards the end of the degree. There is also the assumption that an assessment works at one particular level. A danger with this is that becomes normative, and that it is used as a "quality measure" – the higher the Bloom rating, the better the course.

Johnson and Fuller [27] report on two studies of computer science courses carried out by students in the first year of computer science studies within a university: a panel of assessments rated by instructors, and interviews with the instructors on each course. A significant conclusion from these studies is that the most significant level for many of the courses studied is the *application* level; applying techniques to the creation of artefacts would seem to be at the core of what the study of computing is about. However, for complex application problems students need to use skills that would be classified at the analysis/synthesis/evaluation levels. The authors propose a new level of "higher application" for subjects such as computing. This encompasses cognitive activity that is aimed a solving a problem, yet which needs the traditionally "higher level" skills that engage students at the analysis/synthesis/evaluation level.

A recent paper by Kramer [31] identifies *abstraction* as a core skill that is important for many areas of computer science. The author discusses Piaget's model of cognitive development, which consists of four stages: sensorimotor, pre-operational, concrete operational, and formal operational [54]. His argument is based on studies that show that a significant percentage of the general population do not develop this final stage in the taxonomy: they do not progress to the stage of making significant use of the formal operational processes. Following on from this, he argues that getting students to this stage is a prerequisite for the students studying many aspects of computing, and that we should devise courses that ensure that students reach this stage of general cognitive engagement with material that they encounter *before* teaching most computing topics, or that we could use measures of abstraction ability as a way of selecting students for computing courses.

Finally, Rademacher [56] reports research in progress includes the conceptual development of a model and metrics to determine and classify the level of cognition and added value included in selected knowledge management (KM) systems. He joins Bloom's Taxonomy of Cognitive Objectives and Greenwood's Six C's of the Knowledge Supply Chain in order to contribute a new approach for assessing the role of knowledge management systems including value, skill sets, learning, modeling, and media.

3.1.2 Design of Teaching Materials and Assessments

Another way in which these taxonomies are used is in designing teaching materials and assessments. For example, structuring materials to help students to move through a taxonomy, or structuring assessments so that they assess a wide range of levels of engagement with this material.

A number of authors have discussed how learning taxonomies can be used for assessment design. Lister [38] notes that typical assessments in introductory programming leap straight into higher levels of Bloom's taxonomy, and presents a course design and examples of assessments that move students through the Bloom hierarchy. Thompson [70] reports on the use of the SOLO taxonomy to structure the marking scheme for a programming course, and in particular using this taxonomy to help students understand the grade that they have been assigned. Farthing et al. [19] discuss the design of a new kind of multiple-choice question (*permutational* MCQs) that can be used more readily than traditional questions to assess higher-level skills.

Lahtinen and Ahoniemi [36] are concerned with the use of taxonomies for the design of visualizations to help students understand programming not only in the elementary cognitive levels but to support their progress further also. They look at each level of Bloom's taxonomy, and discuss the kinds of visual material that would be relevant to presenting and interacting with material at each level resulting into a categorization of program visualization examples. Naps et al. [49] make a comprehensive study about the educational effectiveness of visualizations for computer programming education. They identify a set of good practices that have proved to be educationally effective. Bloom's taxonomy is proposed as a standard framework that educators can use to measure such effectiveness. Ihantola et al. [25] have developed a taxonomy of algorithm visualizations: whilst not a "learning taxonomy" as such (it does not give a structure for how students' development is meant to be guided by these visualizations) it could be used alongside such a learning taxonomy to investigate the match between students' development as learners and the technology required to support that development.

Some authors have designed software tools to assist at some level. Thus, Kumar [34] has developed a set of applets (named "problets") to assist at the application level for well-delimited topics. Each problet allows randomly generating instances of a problem involving a concept, a question to be answered and some kind of visualization or interaction to help solving the problem.

Buck and Stucki [11] extend the JKarelRobot environment to give support to all the levels in Bloom's taxonomy. For instance, students are continuously asked the next statement to be executed by Karel. At the end of the run, they are given a score that shows their competence at the comprehension level. Ala-Mutka [2] reports a different automated assessment approach. Facts: there exist different objectives and evaluations but these

objectives are not reached. The reasons are: There aren't obvious and joint criteria and the design of tasks is not careful. A possible solution is to design the objectives, the tasks and the assessments with some obvious criteria based in Bloom's Taxonomy.

Hernán-Losada et al. [21] describe insecurities and ambiguities that they found in applying taxonomies to the design of educational tools. They may classify difficulties into two classes: terminology and the inherent complexity of programming itself. They propose a guide to use the taxonomy within the Computer Science. Moving on from this, in their more recent paper [22] they describe their experiences with designing and developing learning tools inspired by the taxonomy of Bloom. They present a generic framework for the design of these applications and describe the tools developed for the learning of object-oriented programming.

3.1.3 Analysis of Student Responses to Exercises, and Measuring Student Progress

Whalley et al. [72] investigate the results of applying the Bloom and SOLO taxonomies to analysing the results of a programming exercise that was carried out by students at a number of universities. Nine of the questions in this exercise were multiple choice, the final was a free-text question that required students to give an English description of a piece of code. The conclusions of this paper are that the difficulty of these questions correlates strongly with their placement on the taxonomies (in that most students can tackle the lower-rated questions, a subset of those can perform on the higher level questions, then a subset of them on the highest). A particular item of interest is the free-text question that was asked at the end. The authors use SOLO to analyse the responses to these questions. This is carried further in [43] where they analyse the responses to this question and to a further question, related to classifying programs and investigating similarity between programs, and examine students responses using the SOLO taxonomy.

Lister et al. [42] report on the authors use of the SOLO taxonomy to describe differences in the way students and educators solve small code reading exercises. Data was collected in the form of written and think-aloud responses from students (novices) and educators (experts), using exam questions. During analysis, the responses were mapped to the different levels of the SOLO taxonomy. From think-aloud responses, the authors found that educators tended to manifest a SOLO relational response on small reading problems, whereas students tended to manifest a multistructural response. These results are consistent with the literature on the psychology of programming, but the work in this paper extends on these findings by analyzing the design of exam questions.

Lister and Leaney [39,40] also notice that typical programming assignments correspond to level 5 (synthesis). Instead, they group the six levels of the taxonomy into three pairs, so that achieving a level in a given pair yields the corresponding A, B or C grade. In addition, they identify grading practices adequate to each pair, namely lab exercises and exams, multiple choice exams, assignments, projects, and peer review. These ideas have been applied by Box [9], in particular emphasizing the way in which taxonomies can be used to provide a transparent means by which assignments can be explained to students and students can understand their grade and how performance fits into overall progress on courses. In particular, this paper gives comprehensive guidance to lecturers who are considering using Bloom-style structuring for their assessments. Cukierman and McGee Thompson [16] report on the use of Bloom's taxonomy directly with students, in order to help students devise learning strategies to help with their learning of topics in computer science.

The paper by Burgess [13] reports on the author's experience with using Bloom's taxonomy in marking assessments. The grade given to an assessment depends on the level in Bloom's taxonomy that the student's response suggests that that student is working at.

Buckley and Exton [12] review Bloom's taxonomy as a richer descriptive framework for programmers' knowledge of code and illustrates how various software maintenance tasks map to knowledge levels in this hierarchy. A pilot study (with 2 students) is presented showing how participants' knowledge of software may differ at various levels of this hierarchy.

4. EXAMPLES OF THE USE TAXONOMIES IN SOME CANONICAL COMPUTER SCIENCE COURSES

The interaction between typical computing learning outcomes and taxonomies can be further illustrated through examples. This subsection presents three such examples, chosen to be typical of courses that appear in a wide range of computing curricula. One is a first year course, the second is from material that is often given at an intermediate level and the third demonstrates features of final year courses. They are all based on actual courses but have been adapted to suit the needs of this paper. The discussion covers the use of Bloom's taxonomy of the cognitive domain and the SOLO taxonomy, because these are the only ones that we found being used in practice in the computer science education literature. In addition, there is some consideration of Bloom's taxonomy of the affective domain because this could improve constructive alignment between the values instructors want to instill and the ways we assess computing students.

4.1 Introductory Programming Example

4.1.1 Description of course

This is typical introductory object-oriented programming course. It lasts for a single semester and takes an objects-first approach to teaching Java programming, closely following a well-known textbook. The students have lectures and classes (labs) each week. The lectures, which are optional, introduce new concepts. Students are expected to do programming exercises in the class sessions and finish these off in their own time. Some of the class exercises are marked and these marks contribute 20% of the final course result. The main assessment for the course is currently a closed book examination that contains a mixture of multiple choice questions and essay answers.

4.1.2 Learning Outcomes

At the end of the course students will be able to

- Use an object-oriented programming language to write programs.

- Discuss the quality of solutions through consideration of issues such as encapsulation, cohesion and coupling.
- Recognise and be guided by social, professional and ethical issues and guidelines

4.1.3 Assessment using Bloom in the cognitive domain

Bloom's taxonomy in the cognitive domain is conventionally used to assess the first two learning outcomes given above. A typical approach is to write assessment items that are intended to assess at a single level and then to award some fraction of the total number of marks available, depending on how complete the student's response is seen as being. It is relatively unusual to have tasks that are seen as giving students the chance to respond at more than one level, along with assessment criteria indicating which level the student is seen as operating at.

4.1.3.1 Example 1

Consider the following class definition.

```
public class Car
{
        public int numberOfSeats;
        private String model;
        private int engineCode;
        public Car(String model)
        {
                model = model;
        }
        public int getSeats()
        {
                return numberOfSeats;
        }

        private String getModel()
        {
                return model;
        }

        public void setEngineCode(int code)
        {
                int n = code * 2;
                if(code >= 100) {
                        engineCode = n;
                }
                else {
                        engineCode = code;
                }
        }
        }
```

Decide which statement is correct (A, B or C). Only one statement is correct.

Accessors / mutators

(a) The method `getSeats` is an accessor method.

(b) The method `getSeats` is a mutator method.

(c) The method `getSeats` is both an accessor and a mutator method.

Discussion This test item could be assessing recall if it is using an example that the students have seen before. If they have not, it could be a simple example of application of a rule. A conscientious, or over anxious, student could have come across this example before even if it was not used in lectures. A student who did not bother to go to lectures may be applying thise rule from first principles, even if the examiner expects students to be using recall. It is thus hard in practice to determine which of these two Bloom cognitive levels a student is performing at.

4.1.3.2 Example 2

In designing an application, the concept of coupling is important. One guideline states that you should have weak coupling. What is coupling, and why should you have weak coupling?

Discussion This tests whether students have reached the "explain" level. In the unlikely event that the reasons for weak coupling have not been spelt out in lectures, it could be at a considerably higher level.

4.1.3.3 Example 3

Write a method to calculate the winnings of a lottery ticket with three integers, **`a`**, **`b`** and **`c`** on it. The header of the method is

```
public int lotteryTicket(int a, int b, int c)
```

If the numbers are all different from each other, the method returns 0. If all of the numbers are the same, the method returns 20. If two of the numbers are the same, the method returns 10. For example:

lotteryTicket(1, 2, 3) → 0

lotteryTicket(2, 2, 2) → 20

lotteryTicket(1, 1, 2) → 10

Write a full implementation of this method.

Discussion The instructor is likely to expect this to be straightforward example of apply.

4.1.4 Assessment using SOLO

Example 1 above focuses on a single piece of information, ie recognizing the naming of an accessor method. This means that it can be used to assess at the unistructural level.

4.1.4.1 Example 4

Provide two examples of loop constructs that can be used in a method to calculate the minimum value in an array. The header of the method is

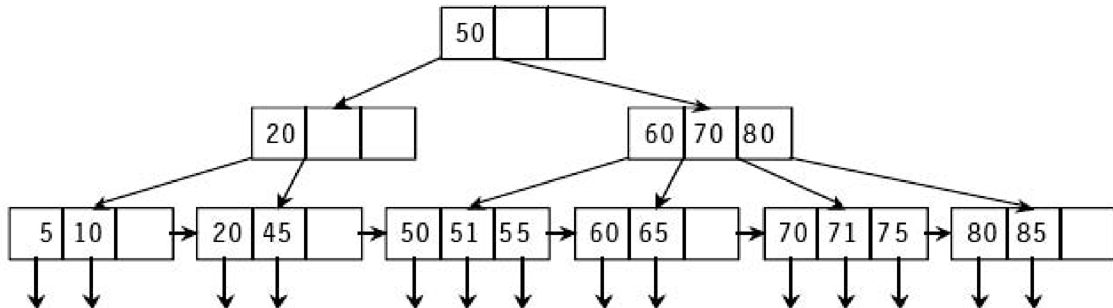
```
public int min(int []a)
```

Discussion this test item requires identification of two distinct loop constructs but not necessarily working code. This means that it can be used to assess at the multistructural level. If the question asked the student to write a routine that calculates the minimum value then it is targeting a relational response since to develop working code requires an understanding of how different constructs work together.

4.1.4.2 Example5

In plain English, explain what the following

For each of the following modifications, show the result B^+-tree obtained by applying the modification to the B^+-tree shown below. Suppose that the maximum fan-out is 4. (Always start with the B^+-tree shown below; do not apply the modifications to the result of previous modifications.)



(a) Insert 21.
(b) Delete 50.
(c) Insert 79.
(d) Delete 10.

Figure 1. Example 8.

segment of code does:

```
bool bValid = true;
    for (int i = 0; i < iMAX-1; i++)
    {
       if (iNumbers[i] > iNumbers[i+1])
       {
           bValid = false;
       }
    }
```

Discussion This seeks a relational response in the sense that the student needs to recognise what is being performed as a whole (relational response) rather than describing the actions of the individual statements (multistructural response); see [40,70,72].

4.1.4.3 Example 6

Performance at the extended abstract level requires students to generalise their knowledge. An example of testing for this could be as follows: the students have been taught how to use an ArrayList. They are now asked to implement code using the Java library LinkedList class. This expects them to generalise the knowledge of working with one collection type and apply it in a near context.

4.1.5 Assessment in the Affective Domain

The learning outcome “Recognise and be guided by social, professional and ethical issues and guidelines” represents an area of learning in which instructors want students to take what they have learnt to heart, not simply to be able to play back what has been told to them. To provide constructive alignment between learning outcome and assessment, it is necessary to assess in the affective domain. The problem is that there is no time for this learning to be embedded, so it is not very feasible to assess it during this module. The answer may be to move the assessment of the affective dimension to a later course.

4.2 Databases Example

4.2.1 Description of course

This course is an introduction to the principles, use, and applications of database systems. It assumes no previous knowledge of databases or database technologies. Topics include: an introduction to relational database systems, relational database model, entity-relationship model, relational algebra, SQL, relational design, and advanced topics such as relational query evaluation, XML databases, and fundamentals of transactions and concurrency.

4.2.2 Learning Outcomes

This course contributes to the development of the following capabilities:

- Enabling Knowledge: Fundamental database concepts including analyzing, designing, defining, constructing and manipulating relational database systems.
- Problem Solving: Ability to design and implement database solutions for various application areas and to build queries for users’ needs, based on analysis of data modeling problem specifications.
- Critical Analysis: Ability to analyze data modeling problem specifications and derive alternative conceptual models that represent the problem in different perspectives leading to alternative database designs.

4.2.3 *Assessment using Bloom's taxonomy in the cognitive domain*

4.2.3.1 *Example 7*

The INSERT statement provides an optional clause to list the columns that you are inserting values into. Why is it prudent to list the columns when you are developing code for a production system?

Discussion This invites students to describe the syntax of an INSERT statement and infer what can go wrong. This is the Comprehension level of Bloom's cognitive taxonomy. However, students who do not know the syntax but have learnt by trial and error that not listing the columns can produce unexpected results may answer at the lower level of Remember.

4.2.3.2 *Example 8*

See Figure 1.

Discussion Most database courses drill students on this kind of problem, so the question requires Application of known rules. Note that no explanation is required. Many instructors would consider that this would make the question more difficult, even though Comprehension, and Explaining in Anderson et al's revision of Bloom's taxonomy, comes at a lower level than Application.

4.2.3.3 *Example 9*

A database contains the following tables:

MOVIE(movieID, title, yearReleased, genre, ratingCode, nationality)

RATING(ratingCode, ratingDescription)

PERSON(name, DoB)

MOVIE_PERSON(movieID, name, role)

where role can take the values "Director", "Producer", etc.

Write a query to return the title, rating, and year released of all movies released from 1970 – 1995 inclusive that were directed by Quentin Tarantino, Ron Howard, or Brian DePalma. Movies should be listed from most to least recent with titles listed alphabetically for each year.

Discussion This type of question typically presents a new scenario to the students, so they are expected to operate at the Analysis level to solve it.

4.2.3.4 *Example 10*

Roger Ebert, a well-known movie critic, wants to compare directors across ratings and genres to see if there are any trends (e.g., do certain directors typically choose movies from a particular genre with particular ratings?). Using the tables in example 10 above, write a query to help Roger analyze the directors who have released one or more movies since 1960. Specifically, list each director along with the genre, rating description, and the number of movies the director has directed in the given genre with the given rating. However, keep the amount of data manageable by only including rows with more than 10 movies. List your results from highest to lowest number of movies. If multiple rows have the same number of movies then list the director, genre, and rating description alphabetically.

Discussion This is a more complex example of analysis. It falls short of Synthesis, or Creating in the revised Bloom's taxonomy, because the problem is very self contained and there is effectively a single right answer. If the student had to find out about the world of movies as well as about databases, it would required synthesis.

4.2.3.5 *Example 11*

For each schedule below, tell whether it is conflict-serializable. If yes, also tell:

- Whether it is recoverable;
- Whether it avoids cascading rollbacks;
- Whether it is possible under *strict* 2PL.

(a) $T1$.write(B), $T2$.read(A), $T2$.write(A), $T1$.read(A), $T1$.write(A), $T1$.commit, $T2$.commit

(b) $T1$.write(B), $T2$.read(A), $T2$.write(A), $T1$.read(A), $T1$.write(A), $T2$.commit, $T1$.commit

(c) $T1$.write(B), $T2$.read(A), $T2$.write(A), $T2$.commit, $T1$.read(A), $T1$.write(A), $T1$.commit

(d) $T1$.write(B), $T2$.read(A), $T1$.read(A), $T2$.write(A), $T1$.write(A), $T2$.commit, $T1$.commit

(e) $T2$.write(B), $T2$.read(A), $T2$.write(A), $T1$.write(B), $T2$.commit, $T1$.read(A), $T1$.commit

Discussion This also requires analysis but falls short of evaluation.

4.2.4 *Assessment using SOLO*

Example 7 above seeks a unistructural response because it deals with a single construct. Example 8 is multistructural because knowledge of both Insert and Delete constructs is required but they are used independently. Examples 9 and 10 target a relational response because the student has to understand how SQL syntax can be applied to her or his analysis of the problem. Example 11 is also seeking a relational response.

4.3 Computing Professionalism Examples

Professionalism within computing is a topic of concern to many professional organizations (IEEE/ACM, BCS etc). These organizations have sought to make professionalism an explicit learning objective (instructional modules) at the university-level. Within computing, this often involves some form of work-based learning. The question of concern is how to assess professionalism? Instructors have often relied on written reports to assess the student's ability to *apply* professional concepts. Additionally, many instructors have attempted to assess professionalism through the use of peer, employee and self evaluations.

4.3.1 *Description of course*

A course in computing professionalism covering topics concerning the social impact, implications and effects of computers on society, and the responsibilities of computer professionals in directing the emerging technology. Relevant professional skills are explored via active-learning activities such as business writing, oral presentations, debates, job hunting and interviewing, professional etiquette, critical thinking, and peer reviewing. An extension to this course gives students the opportunity to apply their skills in consulting capacity, working with real clients to solve their problems.

4.3.2 *Learning Outcomes*

Students completing this course should be able to...

- review and analyze the effects—both anticipated and observed—of the insertion of computer technology into many aspects of society;
- combine their understanding of technology's effects with their personal values, to express and carry out ethical behavior with respect to computing and its impacts, including an ability to articulate and weigh the pros and cons associated with diverse ethical positions;
- identify, analyze, and act upon work situations that have potential ethical, legal, or other professional implications;
- produce written documents of varying type and size in a competent and professional fashion, including the ability to review and critique colleagues' work;
- design and deliver an interesting, concise, and relevant oral presentation with technical content.

Students completing the extended course will

- Be able to apply the concepts and techniques required to build software systems to meet the needs of small enterprises
- Have developed their own computing professional identity through applying the ACM/IEEE code of ethics
- Interact "professionally" with a client through meetings, written reports and email.

4.3.3 Assessment using Bloom's taxonomy in the cognitive domain

4.3.3.1 Example 12 ***A review of a technical article***

Following reviewing and editing guidelines, students are asked to analyze and critique an assigned article, including providing an answer to questions dealing with the organization and writing style of the article.

Discussion This requires students to Evaluate in the cognitive domain. There is also an element of Synthesis (Creating in the revised taxonomy), particularly if the students are expected to extend the review to their own discussion of the topic of the article.

4.3.3.2 Example 13 ***Group Debates***

The debates are intended to sharpen the student's skills to adopt and support one or more viewpoints on an issue about ethics or professionalism in the workplace. The class is broken down into groups of 4-5 students. Each team will choose an ethics topic and write a scenario that raises issues associated with this topic. Teams are instructed to choose topics that have believable arguments both pro and con.

General topics to consider include Special needs, ADA requirements, Universal accessibility, Consideration of public risks in system development, Internet censorship, Competitive intelligence or industrial espionage, Intellectual rights, copyrights, & patents, Privacy, National missile defense system, Protection of the environment or ecology, Ethics of medicine or biotechnology, Scientific fraud or plagiarism, Hackers, Professional and legal liability for defective information or software, Viruses, worms, and other "malware", Technological obsolescence (losing jobs to automation), Cryptography and public encryption, Whistle-blowing.

Discussion This allows students to demonstrate skills of analysis, synthesis and evaluation. Note that because they are asked to take a stance for the sake of debate, they cannot be assessed in the affective domain.

4.3.3.3 Example 15

Proposal to a hypothetical work group about a professional issue: This assignment takes place in four phases. The first deliverable is a two-page (500 words) plan for how the student is approaching the proposal-writing process. The second will be a first draft of an 8-page proposal (approximately 2000 words) researched and written according to the earlier plan. The third deliverable is review of another student's proposal. The fourth deliverable is a final draft of the proposal, in which the student makes revisions and responds explicitly to the review feedback.

Discussion This gives students excellent opportunities to demonstrate synthesis and evaluation.

4.3.4 Assessment using the SOLO taxonomy

If the SOLO taxonomy is used in assessing professionalism then for a unistructural assessment, a single professionalism attribute would be assessed. A multiustructural assessment would seek to assess to professionalism attributes in a way that was independent of each other. A relational assessment would focus on how the professionalism attributes are integrated together in the assessment exercise. An extended abstract assessment would seek to observe professional attributes that are being interpreted in new ways.

In utilizing the SOLO taxonomy, it is not simply the professionalism attributes that can be assessed. In assessing at the relational or extended abstract level, it is possible to assess how professionalism interacts with or relates to other more technical attributes.

4.3.5 Assessment in the Affective Domain

The learning outcomes of the extended course described above are concerned with the development of professional attitudes and values as well as with cognitive skills. These can be measured using a variety of instruments. One is a reflective log, in which students are asked to report their feelings and motives and to evaluate their own performance in the consultancy role. Another instrument is the instructor's observation: was the student proactive in working professionally or was nagging required to ensure that tasks were completed punctually and to a high standard? Finally, feedback from the clients has an important role in determining whether the student's professional values and commitment are demonstrated under all circumstances.

5. WHAT IS SPECIFIC ABOUT COMPUTER SCIENCE

The learning taxonomies discussed in sections 2 and 3 are generic, implying that the types of learning and the ordering of the hierarchy are constant across subjects. However, this may not be the case. For example, in applied subjects such as computing, a principal learning objective is the ability to

develop artifacts (in computing, pieces of software) [30]; by contrast, instructors in other subjects (such as English Literature) place more emphasis on skills of critique and less on producing artifacts (such as novels). It could therefore be argued that in applied subjects, Application encompasses Synthesis and Evaluation, rather than being a lower level skill. It is notable that the recent ACM overview of computing curricula [28] refers to *performance competencies* rather than *learning outcomes*, reinforcing the perceived importance of Application.

We can also distinguish between disciplines in which there is an emphasis on learning through *interpreting* and those in which learning is predominantly achieved through *doing*. Economics and Theology could be seen as examples of the former, Dance and Music performance of the latter. This is not to suggest that Economics and Theology do not require their students to do in the sense of repeatedly writing essays; however they are learning about the practice of the subject rather than running an economy or developing a new religion. Computing students are expected to do a lot of learning through doing, whether it is learning about software engineering by developing systems of increasing complexity, learning about networking by implementing protocols or learning about group dynamics by working in teams.

There are several other characteristics that apply specifically to computer science as discipline. First, and perhaps foremost, studying processes and problem solutions is very central to, if not the essence of, computer science. One could say that solving problems and producing an effective and efficient solution is the core goal of a computer science professional. Computer science centrally involves modeling the real world, representing domains of the most varied nature and complexity, representing knowledge in general and dealing with processes and solutions for problems in such domains.

In order to address the complexities of the problems and domains, there is an essential need to abstract and decompose problems into subproblems and modules. Abstraction, modularity and reusage of previous solutions constitute essential abilities needed by any computer science researcher or professional.

Other characteristics of computer science are creativity and openness to novelty, considering that they are inherently related to finding solutions to problems. It is also worthy of notice that computer science is becoming more and more multidisciplinary, and hence professionals and academics need good communication skills not only among themselves but also with experts in other disciplines.

The following list of keywords encompasses what this working group considers to be intrinsic characteristics of computer science. Clearly, a comprehensive learning taxonomy should be useable for assessment of all of them.

Intrinsic characteristics of computer science:

- Problem solving
- Domain modelling
- Knowledge representation
- Efficiency in problem solving
- Abstraction/modularity
- Novelty/creativity
- Categorization
- Communication skills with experts in other domains
- Adoption of good practice in software engineering

This final feature of computing reflects the need to develop professional skills and values. It is not enough that students should know what constitutes good programming style; we want them to have taken this to heart so that they instinctively write elegant code whenever they work on a piece of software, not just when marks are explicitly available for doing so. Similarly, any intended learning outcome relating to the ACM/IEEE or other professional code of conduct ought to go beyond "Knows about the code of conduct". We want students to respond positively to it by internalizing it and making it part of their personal set of moral and ethical principles, so that they automatically behave according to its precepts, even under challenging circumstances

6. A NEW TAXONOMY FOR COMPUTER SCIENCE

In this section we present a new taxonomy designed to be suitable for computer science and engineering, especially for learning programming (in the broadest meaning of the word). We also present a novel way to apply any existing taxonomy which better deals with modularity and increasing levels of abstraction, aspects that typify engineering and computer science in particular.

6.1 Two Dimensional Adaptation of Bloom's Taxonomy – The Matrix Taxonomy

The intent of the proposed taxonomy is to provide a more practicable framework for assessing learner capabilities in computer science and engineering. The immediate target for this work is computer programming, but we feel the taxonomy is applicable to other fields of engineering in which practitioners produce complex systems. It is meant as a partial solution since (among other things) it does not address the affective domain, only indirectly deals with abstraction skills, and incompletely handles structural relationships in the content.

The inspiration for this taxonomy was research [41,73] indicating that comprehension of program code and the ability to produce program code are two semi-independent capabilities. Students who can read programs may not necessarily be able to write programs of their own. And the ability to write program code does not imply the ability to debug it. Robins et al. [63] describe this independent interpretive skill as the ability to distinguish the intended behavior of the program from the actual behavior of the program.

Although a review of the literature reveals a wide range of possible candidates, only Bloom's taxonomy of the cognitive domain appears to be widely used in computer science course and assessment design. Its main strengths are that the levels are reasonably easy to understand and there is a developing literature, reviewed above, on how to use it to devise test items. Thus we felt it would form the most natural basis for our proposed taxonomy.

We used the revised version of Bloom's taxonomy [6] which responded to problems with the linear approach at the higher levels. It provides a level of creation (Higher Application) which requires competency at all the previous levels and one that does not (Create). In order to visualize this distinction and the semi-independent skills of reading and writing program code, our taxonomy employs a two dimensional matrix with an adaptation of Bloom's taxonomy which is presented in Figure 2.

Figure 2. A graphical presentation of the two dimensional adaptation of Bloom's taxonomy.

The dimensions of the matrix represent the two separate ranges of competencies: the ability to understand and interpret an existing product (i.e. program code), and the ability to design and build a new product. Levels related to interpretation are placed on the horizontal axis and levels related to generation are placed on the vertical axis, with the lowest levels at the lower left corner. The names of the levels are from the revised version of Bloom's, as we feel they are sufficiently unambiguous. It is understood that students traverse each axis in strict sequence. For example, it is not possible to begin to do synthesis (Create) until there is some degree of competency through the Apply Level.

6.1.1 Applying the taxonomy – traversing the matrix

The matrix should be especially useful for instructors needing a marking grid for their students. Also it rather clearly illustrates all the different learning paths students may take, as discovered in recent work by Lahtinen [37].

Different students take different "learning paths" in the matrix taxonomy. For instance, when a student learns a new programming concept he first achieves the knowledge of this concept. At that point the student is in the cell (the state of) "none/Remember" shown in Figure 2. If this student continues with learning by imitating a ready example of a program but without deep understanding of the concept, they will achieve the state "Apply/Remember", i.e. applying/trying to apply the concept without real understanding, with trial and error. This behaviour is illustrated in Figure 3. If instead of imitating, the student decides to first find more information on this concept, as from a book, they might proceed to the cell "none/Understand" to the right of the initial cell. This means that the student is not yet able to produce program code, but he might already understand the meaning behind this concept.

A competent practitioner of a concept would be placed in the cell "Create/Evaluate", which means that he is able to perform at all the competency levels in the matrix. This can also be identified as the level Higher Application [27] and can be reached through different paths as shown in Figure 6.

However, there are students who attain only some of the competencies. For instance, *the theoretical students* identified in a cluster analysis study [37] may be placed in the cell "none/Evaluate" which means that they are able to read program code, analyze, and even evaluate it, but cannot yet design a solution or produce program code. This is not the most common pathway for students to follow, but these students have only proceeded in the horizontal direction as shown in Figure 4.

The same study revealed another group, *called the practical students*, who could be placed in the cell "Create/Understand" of the matrix. Being in that cell would indicate the ability to apply and synthesize without the ability to analyse or evaluate even their own program code. This behaviour is illustrated in Figure 5. The problem for these practical students is in not being able to debug their own solutions when they encounter errors.

Figure 3. A student trapped in trial and error approach

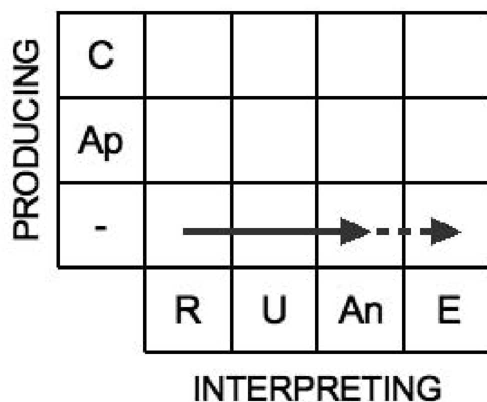


Figure 4. The pathway of the students who attain only theoretical competencies.

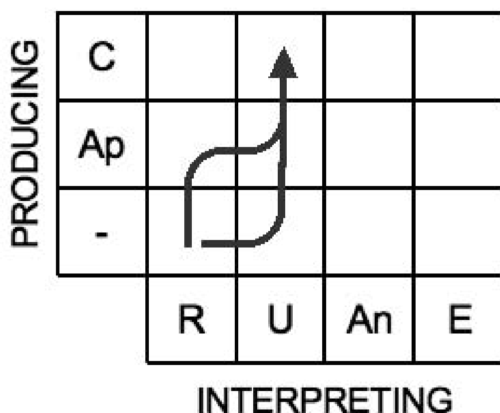


Figure 5. The pathway of the students who attain only practical competencies.

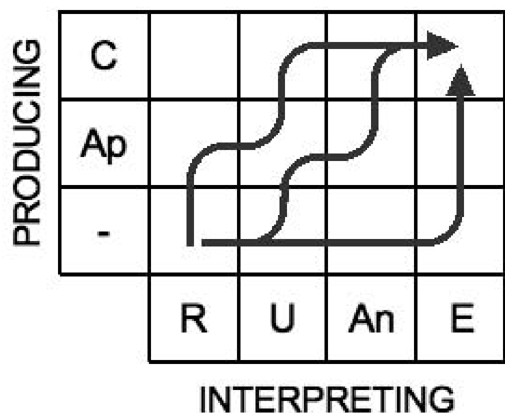


Figure 6. The goal, "Create/Evaluate" or Higher Application, can be reached through different pathways.

Mapping Programming Activities to the Matrix

We provide a mapping from a set of computer programming activities to the cells of the matrix in order to illustrate the discriminatory power of the proposed taxonomy for this subject area. This is done with a list of problem-solving activities related to programming collected as a reaction to difficulties encountered in using Bloom's Taxonomy. The activities shown in Table 1 are mapped to the cells of the taxonomy. See Figure 7.

Table 1 – A list of problem-solving activities related to programming

Solution Activity	**Description**
Adapt	modify a solution for other domains/ranges
Analyse	probe the [time] complexity of a solution
Apply	use a solution as a component in a larger problem
Debug	both detect and correct flaws in a design
Design	devise a solution structure
Implement	put into lowest level, as in coding a solution, given a completed design
Model	illustrate or create an abstraction of a solution
Present	explain a solution to others
Recognize	base knowledge, vocabulary of the domain
Refactor	redesign a solution (as for optimization)
Relate	understand a solution in context of others
Trace	desk-check a solution

To "adapt" a solution probably requires competency close to Create on the vertical scale and at least Understand on the horizontal scale, because modifying involves production and knowing what and how to modify requires understanding. "Apply" in the meaning of Table 1 may be as high as Create on the vertical axis since it calls for some creative ability, probably more than implied by the Apply level, in spite of its name. The position in the horizontal axis depends on the situation. To "debug" calls for a collaboration of both interpretation and building so should be high on both axes, perhaps in the cell "Create/Analyse". The ability to "design" naturally implies Create on the vertical scale and likely some degree of interpretation on the horizontal scale, though how much is uncertain.

"Refactor" and "Relate" are shown at the highest level of interpretation because both call for a deep understanding of the context of the problem and solution. We view "refactoring" as involving an improvement on the original design, thus admitting a possible placement even higher than "design".

To avoid belaboring the mapping example, we simply state that similar reasoning inspired the placement of the remaining activities. The point is that a mapping is feasible and does result in a fairly complete covering of the grid. Furthermore, most of these activities are general enough to be immediately applicable to other fields of engineering.

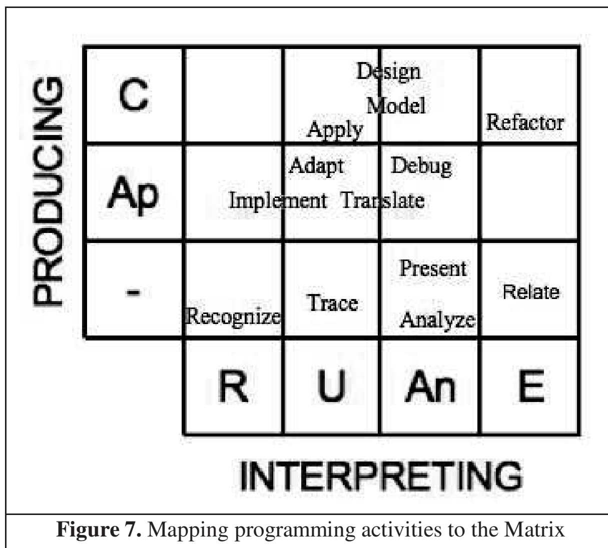


Figure 7. Mapping programming activities to the Matrix

Several of the solution activities may be amenable to assessment using the SOLO taxonomy, which considers the organizational complexity of the problem. This dimension is not at present well illustrated by our matrix, though it may be expected that SOLO levels generally increase as one goes from the origin to the upper right. Consider the activity "present": One would prefer the ability of presentation at the relational level of SOLO as opposed to uni- or multi-structural. "Design", "relate", and "model" are other activities we have identified for which SOLO is useful. In contrast, "implement" as defined in the table, involves applying a process to an otherwise completed design, and thus may be less related to skills involving complexity.

Many of the activities are related to the ability to work with abstraction, an ability that is vital for computer programming and has been discussed as an overriding argument for an alternative learning taxonomy [33]. Design, model, refactor, debug, and present may easily be seen to involve extensive consideration of abstractions. As examples, these activities may include as sub-activities the following: traversing levels of abstraction, mapping between levels (precision being essential for programming!), constructing new abstractions (with the attendant requirements of retaining needed detail and eliminating unneeded detail), adapting abstractions, and using abstractions as models of the original problem and/or solution.

A subject of some discussion in this working group was how to apply the matrix taxonomy to the affective domain. We have designed this taxonomy only for the cognitive domain but non-cognitive skills (e.g. social and emotional skills and the adoption of professional standards) also play a major part in programming practice. Internalization of professional practices is indeed an essential component of learning for computer programmers. Possibilities considered included extending the matrix in one or both directions by another level, or devising a companion matrix. Our overall feeling was that there is so little experience in computer science of assessment of values and attitudes that this would be premature. Krathwohl, Bloom and Masia's taxonomy for the affective domain [32] appears to be usable for courses aiming to develop professional values and we would like to encourage its adoption so that an evidence base can be accumulated.

6.2 Applying Taxonomies Iteratively - a Spiral Architecture for Applying a Learning Taxonomy

Robins at al. describe *a schema* as "a structured chunk of related knowledge" [63]. The student's learning goes through learning new schemas, modifying and combining them in order to produce new, more abstract schemas. Thus, the learning of programming could be seen as an iterative process. In the very beginning, the student is taught really simplistic and basic pieces of information and places to apply them. Instead of learning some things here and there, programming is a skill that is learned by building new information on top of earlier information. So in a way the basic pieces of information students are first struggling with become the bits and pieces they use in subsequent learning of new material. Compared to other cyclic learning styles e.g. the experimental learning style described by Kolb's Learning Cycle [30], the idea here is to proceed to a new level after each cycle.

The idea of a cognitive learning taxonomy can also be used in an iterative, spiral way. When the student is learning the basic concepts and the simplest subjects, he is going through the taxonomy in respect of that subject only. After having created a schema on that subject, he is then guided into a more abstract subject. When looking only at this new subject, the student is starting again from the lowest level of taxonomy—but now using the earlier material as a prerequisite.

The spiral process could be applied to Bloom's taxonomy, in that when the student is learning a new subject, his prerequisites—the materials to use in building new knowledge—have become his new basic knowledge, although the student has perhaps reached the level Create or Evaluate on those earlier subjects. Create could be described as the ability to combine one subject with others in order to build new solutions. This may also be seen when new solutions or subjects are learnt by building upon and integrating previous knowledge. This is easily seen to be true when considering that topics that are difficult and require in-depth analyzing by students are mere basic knowledge for expert programmers. Applying Bloom's taxonomy iteratively is illustrated in Figure 8.

Here is an example of a learning spiral: In the beginning a programming student is taught how to use a loop structure. He will go through all the levels of Bloom's taxonomy while learning it. He *knows* that a loop can be used for iteration; he *understands* how the loop works; he is able to *apply* a loop when told etc., eventually learning it thoroughly. After reaching the highest levels, the loop structure has become a tool for the student to use in subsequent programming. As the student is trying to learn how to sort an array, the loop can be seen as his basis knowledge upon which he is building his new knowledge. Later as the student is trying to implement a top-application[1] to

[1] The application that displays and updates sorted information about the top CPU processes

his own operating system, he will use the sorting of an array as a part of his base knowledge.

Traditionally programming has been taught starting with low levels of abstraction, moving on bit-by-bit to higher abstractions. For example, consider learning expressions, loop structures, functions, classes, design patterns etc. There are still many situations where one returns for more in-depth learning. Using a high level programming language itself establishes a starting level of abstraction, and using the objects-first approach immediately raises that level. The spiral approach with learning taxonomies must not be seen as going directly from bottom to top, but by seeing each round as thoroughly learning some new piece of information which is then used as a basis for the next round in the topic. It is of benefit to know how to write functions using C++ when one is trying to do something similar but more challenging with a lower level language such as Assembly, because then one already has knowledge of procedures, functions, parameters and return values.

The spiral application of a taxonomy is not limited to any particular taxonomy such as Bloom's. One round of the spiral (the learning of a new schema) could be described by any taxonomy suitable for describing students' abilities in that subject. For instance, the Matrix taxonomy proposed in subsection 1 could be applied in a spiral way. One learning path from the elementary level "none/Remember" to the Higher Application level "Create/Evaluate" can be seen as one round of the spiral. When rising to a higher abstraction level, the student starts his " learning path" once again from the lower left corner.

When trying to move up a level of abstraction (as in to start a new round of the spiral) the student may not have reached the Higher Application level "Create/Evaluate". To use his skills as a basic knowledge for the next, more abstract round the student may well be in one of the nearby cells, such as "Create/Analyse". While already progressing in the next round (with a more abstract subject), the student may eventually reach the "Create/Evaluate" state of the earlier level through his experience in using it. Thus the two rounds would in a way be followed in parallel for a while. On the other hand, if the student has taken one of the less desirable learning paths illustrated in Figures 3 and 4 (theoretical or practical only) and attempts to progress to the next round, he could be building his knowledge on misconceptions and may later face problems.

7. CONCLUSIONS AND RECOMMENDATIONS

Despite the wide range of taxonomies presented in this paper the Bloom's taxonomy of the cognitive domain seems to dominate the field of computer science course and assessment design. Though having many benefits, its principal weakness is that the levels do not appear to be well ordered when used to assess practical subjects such as programming. Our recommended solution is to separate Bloom's six levels into two dimensions, Producing (incorporating apply and create) and Interpreting (incorporating remember, understand, analyze and evaluate). This removes the strict ordering while retaining many of the concepts of Bloom's taxonomy. This generates a matrix that can be used to identify a range of different learning trajectories and hence to guide students in how to improve their skills and understanding.

Discussions with colleagues also exposed a lack of alignment between learning outcomes and assessment practice in the area of professionalism. Instructors bemoan students' lack of commitment to good engineering principles but fail to assess this, sending mixed messages to learners. This can be addressed by assessment in the affective as well as the cognitive domain. There no evidence in the literature of this being done, so the most sensible course would be to use an existing taxonomy for this purpose.

We recommend the use of our matrix taxonomy for the design and assessment of programming and software engineering courses. We also recommend that instructors and course designers use Bloom's taxonomy of the affective domain to achieve constructive alignment between their desire to produce computer scientists with professional attitudes and values and the messages they send through assessment tasks. Further work is needed to evaluate both these methodologies in computer science education.

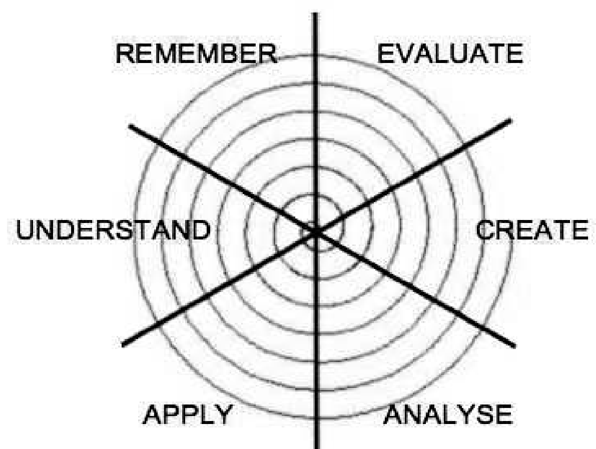


Figure 8. Bloom's Taxonomy as a Spiral Taxonomy.

8. REFERENCES

[1] Agencia Nacional de Evaluación de la Calidad y Acreditación. 2005. Título de Grado en Ingeniería Informática.

[2] Ala-Mutka, K.M. A survey of automated assessment approaches for programming assignments. *Computer Science Education* 15, 83-102, 2005.

[3] Anderson, L.W., Krathwohl, D.R., Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J. and Wittrock, M.C., Eds. 2001. *A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives.* Addison Wesley Longman, Inc.

[4] Azuma, M., Coallier, F. and Garbajosa, J. How to apply the Bloom taxonomy to software engineering. *Software Technology and Engineering Practice: Eleventh Annual International Workshop on,* 19-21 Sept. 2003, 117-122.

[5] Biggs, J.B. and Collis, K.F. 1982. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* Academic Press, New York.

[6] Biggs, J.B. *Teaching for quality learning at university.* Open University Press, Buckingham, 1999.

[7] Bloom, B.S., Engelhart, M.D., Furst, E.J., Hill, W.H. and Krathwohl, D.R. 1956. *Taxonomy of Educational Objectives: Handbook 1 Cognitive Domain.* Longmans, Green and Co Ltd, London.

[8] Bologna Secretariat. Framework of qualifications for the European Higher Education Area, 2005.

[9] Box, I. Assessing the assessment: an empirical study of an information systems development subject. *Proceedings of the fifth Australasian conference on Computing education - Volume 20,* Adelaide, Australia, Australian Computer Society, Inc., 2003.

[10] Buck, D. and Stucki, D. J. Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development. *31st SIGCSE Technical Symposium on Computer Science Education,* 2000, 75-79.

[11] Buck, D. and Stucki, D.J. JKarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum. *Proceedings of the 32nd SIGCSE Symposium on Computer Science Education,* ACM Press, New York, NY, 2001, 16-20.

[12] Buckley, J. and Exton, C. A framework for assessing programmers' knowledge of software systems. *Proc. 11th IEEE International Workshop on Program Comprehension, IWPC,* 2003.

[13] Burgess, G.A. Introduction to programming: blooming in America. *J. Comput. Small Coll.* 21, 19-28. 2005.

[14] Computing Accreditation Commission. *Criteria for Accrediting Computing Programs: Effective for Evaluations During the 2006-2007 Accreditation Cycle.* ABET Inc, Baltimore, MD, 2005.

[15] Cooper, S., Cassel, L., Moskal, B., and Cunningham, S. Outcomes-based computer science education *Proceedings of the 36th SIGCSE technical symposium on Computer science education,* ACM Press, St. Louis, Missouri, USA, 2005.

[16] Cukierman, D. and McGee Thompson, D. Learning Strategies Sessions within the Classroom in Computing Science University Courses*Proceedings of WCCCE 2007, 12th Western Canadian Conference on Computing Education,* May 2007.

[17] Doran, Michael V. and Langan, David D. A cognitive-based approach to introductory computer science courses: lesson learned. *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education,* Nashville, Tennessee, United States, ACM Press, 1995.

[18] Facione, P. A. Critical thinking; A statement of expert consensus for purposes of educational assessment and instruction, research findings and recommendations, 1990*, Fullerton ERIC Reports*, ED315.423.

[19] Farthing, D. W., Jones, D. M. and McPhee, D. Permutational multiple-choice questions: an objective and efficient alternative to essay-type examination questions. *Proceedings of the 3rd Conference on Innovation and Technology for Computer Science Education, ITiCSE, 1998,* ACM Press, New York, NY, 1998, 81-85.

[20] Gronlund, N.F. *Measurement and evaluation in teaching.* MacMillan, New York, 1981.

[21] Hernán-Losada, I., Lázaro-Carrascosa, C. and Velázquez-Iturbide, J. Á. On the use of Bloom's taxonomy as a basis to design educational software on programming. *Proceedings of World Conference on Engineering and Technology Education, WCETE 2004,* COPEC, Brazil, 2004, 351-355.

[22] Hernán-Losada, I., Velázquez-Iturbide, J. Á and y Lázaro-Carrascosa, C. A. Programming learning tools based on Bloom's taxonomy: proposal and accomplishments. *Proc. VIII International Symposium of Computers in Education (SIIE 2006),* León, España, Octubre 2006, 2006, 325-334.

[23] Howard, Richard A., Carver, Curtis A. and Lane, William D. Felder's learning styles, Bloom's taxonomy, and the Kolb learning cycle: tying it all together in the CS2 course. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education,* Philadelphia, Pennsylvania, United States, ACM Press, 1996.

[24] Huitt, W. and Hummel, J. 2003. Piaget's theory of cognitive development. *Educational Psychology Interactive*

[25] Ihantola, P., Karavirta, V., Korhonen, A. and Nikander, J. Taxonomy of effortless creation of algorithm visualizations. *Proceedings of the 2005 International Workshop on Computing Education Research, ICER '05,* Seattle, WA, October 01-02, 2005, ACM Press, New York, NY, 2005, 123-133.

[26] Illinois Online Network: Educational Resources, http://www.ion.illinois.edu/resources/tutorials/assessment/bloomtest.asp, Accessed on 19/07/2007, 2007.

[27] Johnson, C. G. and Fuller, U. D. Is Bloom's taxonomy appropriate for computer science? *6th Baltic Sea Conference on Computing Education Koli Calling 2006,* Koli Calling, November 2006, Berglund, A. and Wiggberg, M., Eds. Department of Information Technology, University of Uppsala, Stockholm, 2007, 120-123.

[28] Joint IEEE Computer Society/ACM Task Force on Computing Curricula. 2005. The Overview Report. http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/CC2005-March06Final.pdf, 2005, visited September 2007.

[29] King, O.M. and Kitchener, K.S. 1994. *Developing reflective judgement: understanding and promoting intellectual growth and critical thinking in adolescents and adults.* Jossy-Bass Inc, San Francisco.

[30] Kolb, D. *Experiential Learning: Experience as the Source of Learning and Development.* Prentice-Hall, New York, NY, 1984.

[31] Kramer, J. Is abstraction the key to computing? *Communications of the ACM* 50, 37-42, 2007.

[32] Krathwohl, D.R., Bloom, B.S. and Masia, B.B. 1964. *Taxonomy of educational objectives: the classification of*

educational goals. Handbook Volume 2: Affective domain. McKay, New York.

[33] Krathwohl, D.R. A revision of Bloom's taxonomy: an overview. *Theory into Practice* 41, 212-218, 2002.

[34] Kumar, A.N. Learning programming by solving problems. In *Informatics Curricula and Teaching Methods,* L. Cassel and R.A. REIS, Eds. Kluwer Academic, 29-39, 2003.

[35] Kundratova, M., Turek, I. *Chapters from engineering pedagogy. Educational Objectives* (in Slovak). STU Bratislava, 2001.

[36] Lahtinen, E. and Ahoniemi, T. Visualizations to Support Programming on Different Levels of Cognitive Development. *Proceedings of The Fifth Koli Calling Conference on Computer Science Education,* 2005, 87-94.

[37] Lahtinen, E. A Categorization of Novice Programmers: A Cluster Analysis Study. *Proceedings of the 19th annual Workshop of the Psychology of Programming Interest Group,* Joensuu, Finland, July 2-6, 2007, Sajaniemi,J. and Tukiainen,M., Eds. University of Joensuu Department of Computer Science and Statistics, Joensuu, Finland, 2007, 32-41.

[38] Lister, R. On Blooming First Year Programming, and its Blooming Assessment. *Proceedings of the Australasian Conference on Computing Education*ACM Press, New York, NY, 2000, 158-162.

[39] Lister, R., and Leaney, J. Introductory programming, criterion-referencing, and Bloom. *Proceedings of the 34th SIGCSE technical symposium on Computer science education,* Reno, Nevada, USA, ACM Press, 2003.

[40] Lister, R., and Leaney, J. First year programming: Let all the flowers bloom. *5th Australasian Computer Education Conference,* Adelaide, SA, Australia, 2003.

[41] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J, Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. A multi-national study of reading and tracing skills in novice programmers. *Working group reports from ITiCSE on Innovation and technology in computer science education,* Leeds, United Kingdom, ACM Press, 2004, 119-150.

[42] Lister, R., Simon, B., Thompson, E., and Whalley, J.L. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education,* Bologna, Italy, ACM Press, New York, NY, 2006, 118-122.

[43] Machanick, P. Experience of applying Bloom's Taxonomy in three courses. *Proc. Southern African Computer Lecturers' Association Conference,* Strand, South Africa, June 2000, 2000, 135-144.

[44] Manaris, B. and McCauley, R. Incorporating HCI into the undergraduate curriculum: Bloom's taxonomy meets the CC'01 curricular guidelines. *Frontiers in Education, 2004. FIE 34th Annual Meeting,* 2004, T2H/10-T2H/15.

[45] Merrill, M.D. Lesson segments based on component display theory. In *Instructional design theory,* M.D. Merrill, Ed. Educational Technology Publications, Englewood Cliffs, NJ, 177-212, 1994.

[46] Merrill, M.D. The prescriptive component display theory. In *Instructional design theory,* M.D. Merrill, Ed. Educational Technology Publications, Englewood Cliffs, NJ, 159-176, 1994.

[47] Merrill, M.D. The descriptive component display theory. In *Instructional design theory,* M.D. Merrill, Ed. Educational Technology Publications, Englewood Cliffs, NJ, 111-157, 1004

[48] Moon, J. *How to use level descriptors.* Southern England Consortium for Credit Accumulation and Transfer, 2002.

[49] Naps, T., Cooper, S., Koldehofe, B., Roessling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R.J., Anderson, J., Fleischer, R., Kuittinen, M. and McNally, M. 2003. Evaluating the educational impact of visualization. *ACM SIGCSE Bulletin* 35, 124-136.

[50] Niemierko, B. *Pomiar sprawdzajacy w dydaktyce. Teoria i zastosowania (in Polish).*Panstwowe Wydawnictwo Naukowe, Warszawa, 1990.

[51] Oliver, D., Dobele, T., Greber, M., and Roberts, T. This course has a Bloom Rating of 3.9. *Proceedings of the sixth conference on Australasian computing education - Volume 30,* Dunedin, New Zealand, Australian Computer Society, Inc., 2004.

[52] Perry, W.G.J. *Forms of intellectual and ethical development in the college years: a scheme.* Harcourt Brace Jovanovich College Publishers, Forth Worth, 1968.

[53] Perry, W.G.J. Different worlds in the same classroom. In *Improving learning: new perspectives,* P. Ramsden, Ed. Kogan Page; Nichols Pub. Co, London, New York NY, 145-161, 1988.

[54] Piaget, J. and Inhelder, B. *The Psychology of the Child.* Routledge & Kegan Paul, 1969.

[55] Polanyi, M. 1958. *Personal knowledge: towards a post-critical philosophy.* Routledge and Kegan Paul, Chicago.

[56] Rademacher, R. Applying Bloom's taxonomy of cognition to knowledge management systems. *1999 ACM SIGCPR conference on Computer Personnel Research,* New Orleans, LA, April 8-10, 1999, ACM Press, New York, NY, 1999, 276-278.

[57] Rapaport, W.J. William Perry's scheme of intellectual and ethical development, http://www.cse.buffalo.edu/~rapaport/perry.positions.html.

[58] Reeves, M.F. An Application of Bloom's Taxonomy to the Teaching of Business Ethics. *Journal of Business Ethics* 9, 609-616, 1990.

[59] Reigeluth, C.M. and Stein, F.S. 1983. The elaboration theory of instruction. In *Instructional-design theories and models: an overview of their current status,* C.M. Reigeluth, Ed. Lawrence Erlbaum Associates, Hillsdale, NJ, 338-381.

[60] Reigeluth, C.M., Merrill, M.D. and Bunderson, C.V. 1994. The structure of subject matter content and its instructional design implications. In *Instructional design theory,* M.D.

Merrill, Ed. Educational Technology Publications, Englewood Cliffs, NJ, 59-77.

[61] Reigeluth, C.M., Merrill, M.D., Wilson, B.G. and Spiller, R.T. 1994. The elaboration theory and instruction: a model for sequencing and synthesizing instruction. In *Instructional design theory,* M.D. Merrill, Ed. Educational Technology Publications, Englewood Cliffs, NJ, 79-102.

[62] Reynolds, C. and Fox, C. 1996. Requirements for a computer science curriculum emphasizing information technology: subject area curriculum issues. *ACM SIGCSE Bulletin* 28, 247-251.

[63] Robins, A., Rountree, J. and Rountree, N. 2003. Learning and Teaching Programming: a Review and Discussion. *Computer Science Education* 13, 137-172.

[64] Sanders, I. and Mueller, C. A fundamentals-Based Curriculum for First Year Computer Science. *31st SIGCSE Technical Symposium on Computer Science Education,* ACM Press, 2000, 227-231.

[65] Scott, T. Bloom's taxonomy applied to testing in computer science classes. *J. Comput. Small Coll.* 19, 267-274, 2003.

[66] Simpson, B.J. The classification of educational objectives: psychomotor domain. *Illinois Journal of Home Economics* 10, 110-114, 1966.

[67] Svec, S. Taxonomy for Teaching: A System for Teaching Objectives, Learning Activities and Assessment Tasks (Revision of Bloom's Taxonomy of the Cognitive Domain). In *Pedagogicka revue* (in Slovak) 57, 453-476, 2005.

[68] Swedish Ministry of Higher Education and Research, Higher Education Ordinance, http://www.sweden.gov.se/sb/d/574/a/21541, Accessed on 19/07/2007, 2007.

[69] Thompson, E. Does the sum of the parts equal the whole? *Proceedings of the seventeenth annual conference of the National Advisory Committee on Computing Qualifications,* Mann, S. and Clear, T., Eds. National Advisory Committee on Computing Qualifications, 2004, 440-445.

[70] Thompson, E. Holistic assessment criteria - applying SOLO to programming projects. *Proceedings of the Ninth Australasian Computing Education Conference (ACE 2007),* Ballarat, Victoria, Australia, Mann, S. and Simon, Eds. Australian Computer Society Inc, 2007, 155-162.

[71] University of Victoria. Learning Skills Program - Bloom's Taxonomy, http://www.coun.uvic.ca/learn/program/hndouts/bloom.html, Accessed on 19/07/2007, 2007.

[72] Whalley, J.L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K.A., and Prasad, C. An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52,* Hobart, Australia, Australian Computer Society, Inc, 2006, 243-252.

[73] Winslow, L.E. 1996. Programming Pedagogy - a Psychological Overview. *SIGCSE Bull.* 28, 17-22.

Acknowledgements

This paper is partly supported by the ITiCSE 2007 Student Bursary of the British Computer Society and the grant of FGU No 21/2007 (Faculty of Management Science and Informatics, University of Zilina)

Publication (ii)

Lahtinen, E., Ahoniemi, T. Visualizations to Support Programming on Different Levels of Cognitive Development *In: The Proceedings of The Fifth Koli Calling Conference on Computer Science Education*, pages 87–94, November 2005.

Visualizations to Support Programming on Different Levels of Cognitive Development

Essi Lahtinen
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
essi.lahtinen@tut.fi

Tuukka Ahoniemi
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
tuukka.ahoniemi@tut.fi

ABSTRACT

Most of the visualizations on the level of introductory programming concentrate on presenting programming concepts to the learner. However, the learning problems of novice programmers are not usually in concept understanding but culminate to major difficulties in producing programs on their own. Thus, the needs of the learner and the purpose of the visualizations do not seem to meet.

Bloom's taxonomy introduces six levels of cognitive development that the learner follows. Widening the scope of visualizations to cover all these levels can be a solution for guiding the learning better by visualizations.

This paper introduces a categorization of visualizations. The categories presented are related to the levels of Bloom's taxonomy. To show how the categorization can be used, we also introduce examples and ideas on how to develop visualizations to the categories.

Keywords

Bloom's Taxonomy, Visualization, Programming, Learning, Teaching

1. INTRODUCTION

Visualizations have been used to aid learning in many areas of computer science education but we believe they are not used pedagogically as advantageously as they could be. In the introductory programming course level there are visualizations available for free use. Our work suggests a possible solution on how they could be developed to be more effective and directed better for the needs of the learner.

The biggest problem of novice programmers does not seem to be the understanding of basic programming concepts but rather learning to apply them [5]. It has been concluded [10, 11] that novice programmers are typically limited to superficial knowledge of programs and tend to approach the program 'line by line' rather than looking at the larger wholes of the program. They may know the syntax and semantics of individual concepts, but do not know how to build programs using them [13]. Hence, it is important to emphasize the usage strategies of the concepts in the teaching materials.

In this paper we introduce ideas on how to produce visualizations for novice programmers to enhance the students' own programming skills (not only concept understanding). The ideas for developing visualizations are divided into different levels according to the Bloom's taxonomy of the levels of cognitive development. Section 2 will start with presenting work done by visualization tool developers earlier. In Section 3 we will introduce Bloom's taxonomy and in Section 4 its application in visualizations by presenting a categorization of visualizations. Section 5 will present the examples and ideas for developing the visualizations. Section 6 analyzes the implementation of visualization. Some visualization tools are also placed in the categories as examples. The conclusions are presented in Section 7.

2. RELATED WORK

On the level of introductory programming courses, most of the visualizations concentrate on explaining the program code and its behaviour. There are visualization tools such as VIP [12] for C++ and Jeliot 3 [7] for Java that contain an interpreter of the programming language or its subset and can thus visualize any given structures or little programs. One benefit gained with an interpreter based tool is the easier creation of the actual visualization examples. With the interpreter handling the code and its visualizing the person creating the example can concentrate only on the actual content.

Some introductory programming course level visualizations, e.g. [1], are built one by one to visualize just one certain program, for example with Flash [2]. In these cases, creating the example can take much more time than with the interpreter based tools. Nevertheless, many examples on this level are plain *click-and-watch examples.* Some of the examples also enable the student to actively participate in the visualization by providing his own input for the program. The level of the learner engagement seems to be a critical factor in enhancing the learning with visualizations [8]. Since it is usually easier to make the interpreter based tools interact with the learner, the effort spent in developing the interpreter is usually worthwhile when measuring the pedagogical value of the tool.

On more advanced level of programming, for instance in courses about data structures or algorithms, the visualizations are usually developed to provide a more abstract picture of the visualized structure instead of staying in the code structure level as in the basic programming visualizations. One obvious reason for this is that these subjects are usually not directly associated to only one programming language but rather to pseudo code presentation. Examples of visualization tools on an abstract level are Trakla [6] and JHAVÉ [9] that concentrate on the behaviour of algorithms and their usage on data structures.

Level of the taxonomy	Description
1. Knowledge	The student knows the facts or the concepts from the area, no real understanding
2. Comprehension	The student perceives the meanings of the facts and understands how the concepts work
3. Application	The student is able to apply the knowledge in new situations for example in exercises described by the teacher
4. Analysis	The student can identify and solve problems on his own and analyse the components of the subject
5. Synthesis	The student can make own conclusions from the earlier levels and generalize what he has learnt for new situations and thus build larger/new wholes combining his earlier knowledge
6. Evaluation	The student can compare and assess different kinds of ideas and make choises according to this reasoning

Table 1: Introduction of the levels of Bloom's taxonomy

Many visualizations present concepts without processing them further. Since learning problems are often connected to more advanced issues than individual concepts [5], we suggest that learning materials could be directed to develop program generation, modification and debugging skills. Creating programs on ones own is important in learning programming, so visualizations should have more problem solving nature instead of only representing concepts.

3. BLOOM'S TAXONOMY AND ITS USAGE IN TEACHING PROGRAMMING

The taxonomy developed by Bloom et al. [3] categorizes six different levels of cognitive development when learning a subject. The taxonomy is general and can be applied to studying computer science and programming among others.

The levels of the taxonomy are shown in Table 1. They are often approached linearly, because it is easier to proceed to one of the levels if the learner knows the earlier levels of the taxonomy. On the other hand, some of the levels (e.g. synthesis and evaluation) could also be inverted [4].

Here we discuss the levels using loop structures as an example. The lowest level of the taxonomy is *knowledge*, which means that the student knows the facts or the concepts from the area, for example when learning loop structures in programming, this level could mean that the student knows that `while`, `for` and `do – while` loops exist and their syntax. This does not require understanding of the actual behavior of the loop structures, just the knowledge that they are used for repeating things. With more understanding of how a loop works and the differences between the three loop structures and their behaviour the student has reached the second level, *comprehension*.

In the third level, *application*, the learner already understands the subject so well that he is able to apply it in new situations. For example, he can add a loop structure to an existing program or modify the condition of the loop. When he is able to analyze code for example to recognize the purpose of a complex loop structure or to fix a logically broken loop, the student's level of cognitive development is on stage four, *analysis*. On this level, concepts such as a loop invariant could be introduced to the learner, since they need a more analytical way of looking at the code.

Synthesis covers making ones own conclusions from the earlier levels. It is the fifth level in the taxonomy. On this level the learner is able to use the loop structures in his own program code. Combinement with his earlier knowledge, for example using function calls as the condition of a loop statements, is also placed on this level. At this point the learner also knows in which kind of situations a loop structure is needed.

The sixth and the highest level of the taxonomy is *evaluation* which means that the learner can compare and judge different kinds of ideas in building and using loop structures. As with the second level, *comprehension*, the learner compared the meaning of different loop structures, in level six he is able to compare their adequacy for certain situations and thus design clearer (to understand and to implement) programs using loops.

Naps et al. [8] have shared ideas on what the student can do on different levels of the taxonomy and what kinds of tasks the student can be given to support learning on these stages. Their study is about teaching data structures and algorithms. The taxonomy can be used similarly in introductory programming courses. Our aim is to improve the usage of the taxonomy in visualization development on the area of introductory programming.

4. VISUALIZATION CATEGORIES

We have analyzed the different kinds of visualizations that exist and planned new types of visualizations that could be helpful for students who are learning the basics of programming. We recognized that to achieve the different levels of Bloom's taxonomy the student needs different kinds of materials. Visualizations can be used versatilely for supporting these different levels. Table 2 introduces the different categories of visualizations we have recognized and their relation to the taxonomy.

So far the visualizations used on introductory programming courses mainly focus on the two first levels of Bloom's taxonomy: knowledge and comprehension. With more effective visualization tools and more effort in designing the visualizations the cognitive stage can be deepened to further levels.

4.1 Illustrative visualizations

Visualizations are mostly used presenting profound examples on new subjects to students. We call these *illustrative visualizations* because they do not usually require much participation from the student. The student mainly follows the visualization. Typical characteristics for illustrative visualizations are:

- Brief and simple.

Level of the taxonomy	Visualization category
1. Knowledge	-
2. Comprehension	Illustrative visualizations
3. Application	Utilizing visualizations
4. Analysis	Problem-solving visualizations
5. Synthesis	Productive visualizations
6. Evaluation	Discerning visualizations

Table 2: The application of Bloom's taxonomy in visualizations

- The code is often made just for illustrating a certain language feature and does not necessarily implement a program which could be useful elsewhere.
- Profound instructions describing both the syntax and the semantics.
- Step-by-step program run.

Introducing a new concept superficially reaches only the first level of the taxonomy (*knowledge*). Animating and deeply explaining each step can help student reach level two (*comprehension*). Letting the student lead the run of the visualization himself for example by entering his own input for the program and following the behaviour of the program the understanding can help reaching level two even better.

The reason why we do not introduce a name for a visualization category reaching only the first level of the taxonomy (*knowledge*) is that implementing visualizations for this level only is quite difficult and useless. Let us consider a situation where a novice programmer has learned how to print a text and then wants to repeat printing the same text. The teacher tells him that the problem is solved by implementing a loop and demonstrates the syntax of a loop. At this point the first level, knowledge that loop statements exist, has already been reached. Supporting only this with visualization has no actual use because the same visualization example can give more than just the syntax.

The benefits of illustrative visualizations take place when the student wants to know how something actually works and in what kind of situations it can be used. Being able to answer these questions makes the difference between the taxonomy levels one and two. Thus we suggest that visualizations should be developed to support at least the level two of Bloom's taxonomy.

4.2 Utilizing visualizations

To deepen the cognitive stage of visualizations to the level *application*, the student should participate more than just following the visualization as in the illustrative examples. One solution is to build programming exercises to run on a visualisation tool. We call the next stage of the visualization categories *utilizing visualizations* because in this category the student is supposed to both visualize the code and use the knowledge he has gained so far.

The skill the student has to practise on this level of the taxonomy is to apply his knowledge in a new situation. As a basis for doing this, the student can be given a program that already implements a little programming structure. The task can be to modify the behaviour of this structure to satisfy new requirements or to implement a little extension to the functionality. Since the goal is not to produce a completely new program yet and the task requires only simple knowledge application, this category is called *utilization* instead of *production*.

The visualization tool makes these kinds of examples easier for the student because the student can start with getting to understand the existing code by visualizing it. After understanding the code the student has a basis for making a step forward to make some modifications to the code. Writing his own pieces code for solving the problem makes the student apply the concept in a new situation which is a skill rated on level three (*application*) of Bloom's taxonomy.

4.3 Problem-solving visualizations

On the fourth level of the taxonomy (*analysis*), the student is supposed to identify problems and solve them, which naturally created the name *problem-solving visualizations* for the next category of visualizations. Since problem solving is a common method of teaching, there are numerous different ways of building problem-solving visualizations. We introduce two different subtypes of problem-solving visualizations in the following subsections. Certainly, many more subtypes can be developed.

4.3.1 Debugging visualizations

Maybe the simplest way to produce a problem-solving visualization is to give the student an example program that contains an error and a description what the code should do. Naturally the error cannot be a simple syntax error but a logical one, which can be difficult to locate. The student can first animate the code with the visualization tool to locate the error. This is usually much easier for the student than locating an error by just reading the code. Often the student has to give different inputs to the program and visualize it more than once to find the error.

After tracing the error the student can then try his own solution for fixing the bug and see if the solution was a correct one by visualizing the fixed code. An automated testing for correctness can also be used with feedback and hints if possible. This, of course, requires more from the used visualization tool.

4.3.2 Analysable visualizations

The code example in problem-solving visualizations need not to be broken. The teacher just has to be more creative in defining the problem when the code already implements the desired functionality. At its simplest, the problem can be to understand and explain what does the program implement and how does it reach the final result in detail. The question is not just about the final state of the program, because this can be seen by just running the visualization to the end. The main points the student has to find out are how the program reaches the result and which phases it has to go through. To direct the student to divide the problem into smaller parts and analyze each of them carefully, the assignment can include smaller, more detailed questions.

Naturally examples like this could be done just in paper and pen, but if the student can watch the code animated, it can help him to solve the problem. We believe that aiding with visualization does not lessen the learning task beyond this example, but supports it by providing another perspective to approach it. Nevertheless, going through the code so deeply that the student is able to answer questions about it, deepens the stage of learning and rises up to the level four (*analysis*) of the taxonomy.

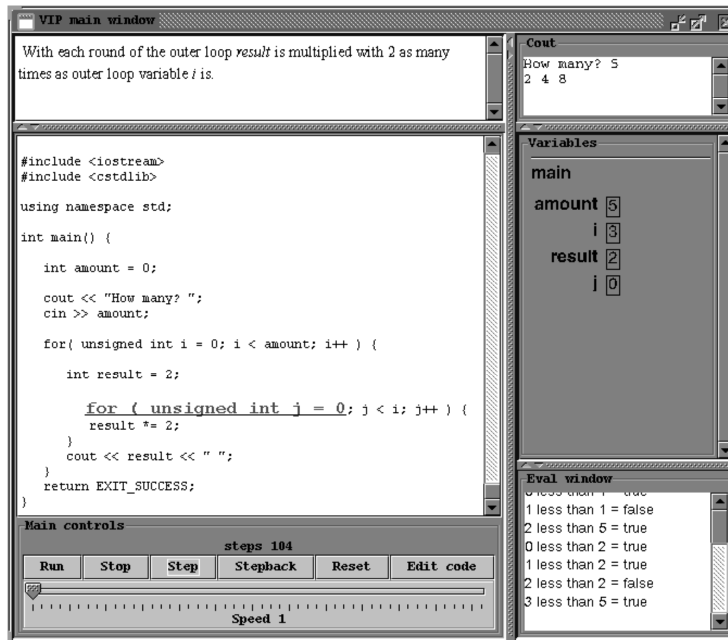


Figure 1: Illustrative visualization of loops. The program is running in the underlined line.

4.4 Productive visualizations

The fifth level of the taxonomy is *synthesis*, which means combining the learner's earlier knowledge to produce new wholes. Thus we handle this level by assignments to implement a completely new program without a code to get started with.

Again, there are different ways of aiding this kind of tasks by visualizations. Here we introduce two subcategories: *visualizing the student's solution* and *visualizing the assignment*. These two subcategories can also be combined so that both of the ideas are used to aid the student in implementing his own program.

4.4.1 Visualizing the student's solution

The interpreter based visualization tools that enable the student to visualize his own code also provide the possibility to start creating a completely new program from a scratch in the visualization tool. This way the student is using the visualization tool to replace his usual program development environment. The teacher can give an assignment to implement a new program that satisfies certain requirements. Then the student can visualize his own solution or parts of it during the programming process.

The normal way of doing programming assignments just using an ordinary programming environment instead of a visualization tool also supports the stage five (*synthesis*). The benefit gained by using a visual environment is to support the students who have problems in piecing together how the program runs. In a way, the usage of the visualization tool replaces the usage of a debugger for finding the places where the program behaves wrong. Using a visualization tool can be more beneficial since novice programmers very seldom know how to use a debugger effectively.

The idea of working on a code in a visualizatin tool is that the student can implement a little part of the program and visualize it before writing the rest of the program. Sometimes visualizing the code already in the very early development stages can reveal mistakes in the thinking process of the novice programmer. The pictures provided by the visualization tool usually help the learner recognize if he has made wrong choises with the data types. When following the execution of the program in the visualization tool, the student often also recognizes when the control of the program does not follow the path he has meant it to follow. When the errors are found, it is also possible to take an action to correct them.

4.4.2 Visualizing the assignment

In bigger programming assignments, visualizations can also be helpful in giving the assignment. In this case, it is not the student's code that is visualized but the assignment itself. Especially the novice programmers have sometimes difficulties in reading a specification that is complicated and full of formal explanations on how the program is supposed to work. Sometimes looking at a visual presentation on the expected program behaviour gives a much clearer picture on what to start working on, even though the student still needs to build the program structures by himself. The point is not giving the assignment only by a visualization but to complete the assignment using a visual presentation.

This kind of visualization of course requires the visualization to show the main points of the algorithm and not just the programming structures. In the earlier types of visualizations, we have concentrated of visualizing the code. For preparing this type of visualizations we need a new kind of tool on a higher level to support visualizing the behaviour of the algorithm.

4.5 Discerning visualizations

It is very important to teach the student to look at the program critically from the beginning of learning program-

ming. This is also required on the level six (*evaluation*) of the taxonomy where the student is supposed to assess programs according to his own reasoning and make choices based on this assessment.

In the more advanced programming courses, different kinds of solutions are often compared, for instance, by looking at the asymptotic time and memory consumption of the algorithm. In the introductory programming courses, the comparison need not to be so formal and the comparison can be done simply using common sense and making one's own conclusions about the behaviour of the program. This can be aided by visualizations. The different solutions can be visualized to give a basis for the student's own reasoning. The differences are easier to be handled and discussed, when the student can see them concretically in the visualization.

5. VISUALIZATION EXAMPLES

We use loop structures as an example of a course content in which learning can be aided by visualizations. Loop structures is a concept introduced quite in the beginning of learning programming regardless, if the approach to programming is imperative or object oriented. Even if introduced in the beginning, it usually is not easy for students, so visualizations are useful in teaching it. All the examples presented in this section handle loop structures.

We cover all the different types of visualizations introduced in Section 4 in our examples. The examples are kept simple, so they can be presented to students on their first programming course and do not require understanding of complex algorithms.

5.1 Illustrative visualizations

Figure 1 shows a screen shot of an illustrative visualization implemented with VIP [12]. The example code shows the use of two nested `for`-loops. The program prints given amount of the powers of two. The program is run step-by-step with instructions at the window in the top-left corner describing the run of the program in detail. Concurrently, the state of the program is animated in *Variables* window on the right side. As a hint for the student, the assignment can propose to try the behaviour of the program also with the input 0. This example is comparable to a lecturer first introducing subject to the students.

5.2 Utilizing visualizations

Figure 2 shows an example where the visualization tool contains a ready made program that prints the five first results of the multiplication table of six. The student can visualize this code to get started. The task is to change the program to print the five first powers of six. Only a small part of the code (with a white background in Figure 2) is left for editing. After a short reflection student finds the task to be implemented with a loop. Solving this problem rises up to the third level of Bloom's taxonomy (*application*) because student must independently notice the need for a loop in the solution and then implement it.

An even easier version of a utilizing visualization could be, for instance, that the given code example contains an infinite loop structure (`while( true ) {...}`) that repeats something useful. The student is supposed to modify the loop by inserting a rational condition for the loop so that the program run will end at some point. Another assignment for the student could be implementing a `for`-loop when the original version in the visualization tool uses a `while`-loop.

```
int counter( string word ) {
  int longest = 0;
  int count = 0;

  for( int index = 0; index < word.length(); ++index ) {
    count = 1;

    if( index != word.length() - 1 ) {
      while( word.at( index ) == word.at( index+count ) ) {
        count++;
      }
    }
    longest = max( count, longest );
  }
  return longest;
}
```

Figure 3: The code example of a problem-solving visualization (debugging visualization) on loops

The modification tasks can be built so that the student is able to modify all the program code or so that only some parts of the code are left for editing. This, naturally, depends on the visualization tool and its capabilities. Sometimes with the novice students, it can be clearer to point for the student exactly which part is he supposed to edit.

5.3 Problem-solving visualizations

In Section 4 we presented *debugging visualizations* and *analysable visualizations* as examples of *problem-solving visualizations*. Here we give an example of both of them.

5.3.1 Debugging visualizations

Figure 3 presents a code example that could be used for implementing a *debugging visualization*. The code example includes the definition of function *counter* that counts the longest sequence of same characters following each other in a word. The code contains an index out of bounds error that occurs only if the word ends with multiple similar characters, so it is not easy to find. The task given for the student is to locate the error and correct it.

The correction the student needs to carry out is simply to modify the condition of the loop. On each round of the loop it needs to be checked that the index of the string is not out of bounds. I.e. the student needs to add an extra checking to the condition of loop.

5.3.2 Analysable visualizations

Figure 4 describes an example of an *analysable visualization*. The student is given a complex function (non-helpfully named as `aparator()`) using three nested `for`-loops. The task is simply to analyze what the function does. Some guidance is provided with few smaller questions to begin with.

This kind of a code can easily be visualized by existing visualization tools. The questions presented in the example are open, so checking the correct answers is not possible by software. There could also be modifications of this exercise so that the questions would be multiple choice. Then the students' answers could also be checked by the visualization tool. On the other hand, open questions can be more helpful for the student, because they leave more space for his own reasoning.

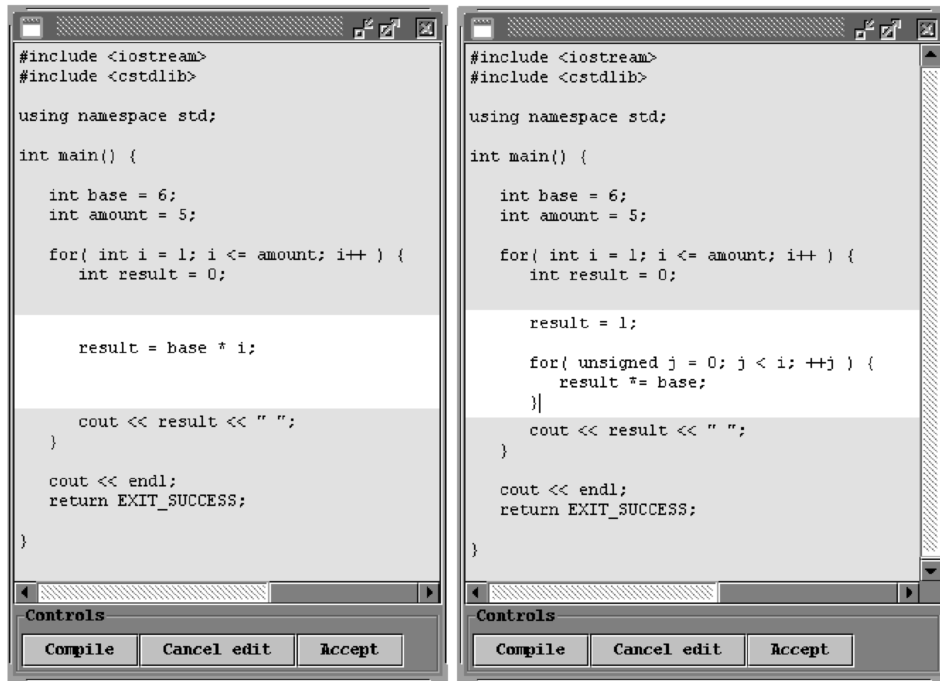


Figure 2: A utilizing visualization on loops.

```
 1 void aparator( vector< vector< int > >& A ) {
 2
 3    for( unsigned int i = 0; i < A.size(); ++i ) {
 4
 5       for( int j = static_cast<int>( A.at(i).size() - 1 );
 6            j >= 0; --j ) {
 7
 8          for( unsigned int k = 0; k < ( i + j ); ++k ) {
 9             A.at(i).at(j) += A.at(i).at(j);
10          }
11       }
12    }
13 }
```

Examine the function `aparator()` and answer the following questions:

1. With a 3x3 array as a parameter, how many times does
   a) the loop starting from line 3 run?
   b) the loop starting from line 5 run?
   c) the control of the program reach line 9?
2. Line 5: Why is `- 1` required?
3. Line 5: What would happen, if variable `j` was unsigned like variable `i`? Why?
4. What does `aparator()` do?

Figure 4: A problem-solving visualization (analysable) on loops.

5.4 Productive visualizations

As the name of this category tells, on this level the student is producing a program on his own. We do not necessary need an example program for the student to start with. The task can be given as a simple specification of a program behaviour.

5.4.1 Visualizing the student's solution

As an example of a visualization tool that could be used as a program development environment we mention VIP [12]. It contains a possibility for the student to enter his own code and visualize its behaviour. VIP's code editor was built specially for this kind of usage.

Any programming assignment that requires the student to use loop structures and combine using them with his earlier knowledge will be good for this category. The most important point is that the assignment requires the student to completely design and implement the program without an example to follow or a template to start with. One example could be to implement a program that sorts the content of an array since the simplest sorting algorithms require repeating things with loop structures.

5.4.2 Visualizing the assignment

As already suggested in the previous subsection, after learning how loop structures work, the student could implement a very simple sorting algorithm, e.g. selection sort. This, however, is a very challenging task for a novice programmer, but it can be aided very much by adding a visualization whose main points are described in Figure 5 to the textual assignment description.

This way the algorithm is given to the student and he can concentrate on implementing the right kind of loop structures to make the algorithm work on the programming lan-

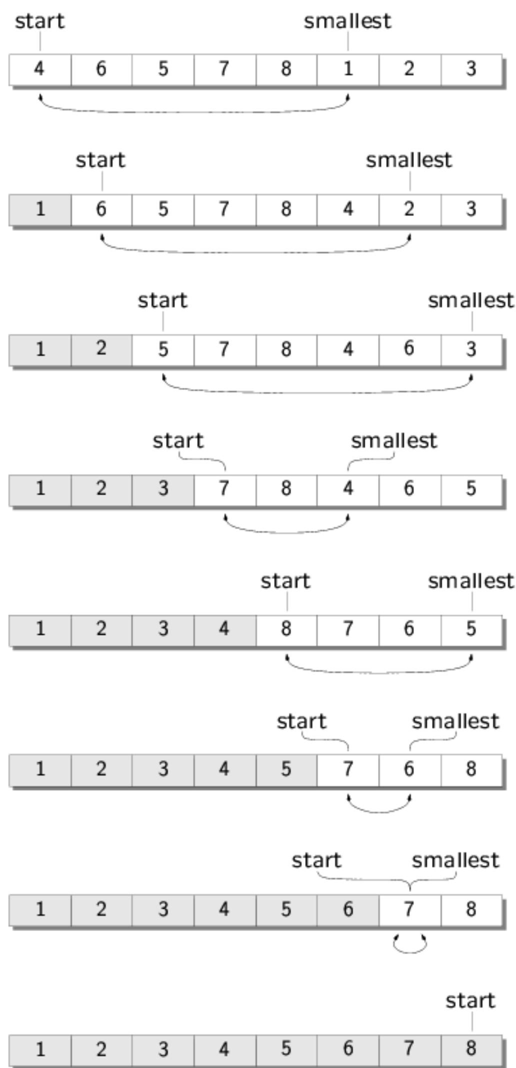


Figure 5: An example on how to explain selection sort clearly using a visualization.

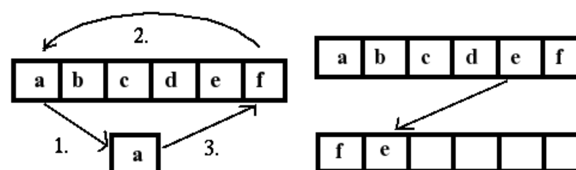


Figure 6: Two different ways to reverse a string.

guage. At some point of the studies, it is essential to be able to design the algorithms too, but even in the beginning, it is very educative to implement a given algorithm.

5.5 Discerning visualizations

As an example case of using visualizations to compare different kinds of solutions on the same subject, we use reversing a string. It is a simple programming task that can be solved in many different ways. Figure 6 demonstrates two different ways to reverse a string. The demonstration is naturally much more understandable as a visualized presentation than as a still picture like here.

When just looking at the different program codes reversing a string it is quite obvious that they are both loop structures, and it can be very difficult for a novice programmer to say much more about it. The animated presentation in the visualization tool can concretize the differences so much that the student can do assessment that will reach the level six of the taxonomy (*evaluation*). The student can easily recognize, that solution on the right side (Figure 6) requires extra memory for only one character and the loop only needs to go through half of the string where as in the other solution the need for memory is bigger and the loop needs to handle the whole string.

As another example of a discerning visualization we can consider programs doing the same task in an iterative way and in a recursive way. There are many different examples about tasks that can be solved both ways.

Building *discerning visualizations* can be easier for the teacher if he collects frequently given answers to problems from the students and chooses the best ones for the visualized examples. One way to make the student even more engaged in the evaluation is to let him solve the problem independently first. Then he can look at the different solutions as visualizations and discuss the pros and cons of his own solution compared to the other possibilities.

This kind of visualizations can be done in a visualization tool that concentrates on animating the code structures. It can, however, be more beneficial by a visualization that stays on a more abstract level showing the main principals of the algorithm.

6. IMPLEMENTATION OF THE VISUALIZATIONS OF DIFFERENT CATEGORIES

As already stated, most of the visualizations available on the field of introductory programming are *illustrative visualizations* (level 2). On the level of learning data structures and algorithms, Trakla provides *problem-solving visualizations* (level 4). The *problem-solving visualization* examples of data structures are different from the ones presented in this paper even though the idea behind them is the same according to Bloom's taxonomy.

Visualization tools that allow the code (or rather desired parts of it) to be edited (Jeliot 3 [7] and VIP [12] for instance), enable developing visualizations to the categories *utilizing visualizations*, *problem-solving visualizations* and *productive visualizations.* More traditional programming exercises (e.g. on pen and paper) can be mixed with visualization tools providing this feature to produce more supportive teaching (or self-learning) material. Supporting this kind of tasks with visualizations is not possible by tools like Flash [2] that do not "understand" the logic of the visualized program.

In visualizations supporting introductory programming, the categories *illustrative visualizations* and *utilizing visualizations* plus some visualizations in the categories *problem-solving visualizations* and *productive visualizations* can be mainly animated on a tool that shows how the programming structures work on code level. In some cases in the categories *discerning visualizations* and *productive visualizations* the animation needs to show the behaviour of the algorithm on a higher level and the tools built for code visualizations are not enough. In the more advanced programming courses, the visualizations are often used particularly on higher level.

In all the visualization categories except *illustrative visualizations* the efficiency of the visualization can be improved by adding some sort of intelligence to recognize the correct answer and notify the student when it is achieved. This would require more effort from the visualization tool developers, but it would improve the quality of the visualization.

There are no visualization tools available that would support all the visualization categories perfectly. The needs of the different categories are so different that building a tool to support all of them is very challenging. As a consequence of all these features required from the visualization tool implementing one becomes labourious work.

As future work on this subject we are planning to concentrate on how the different visualization categories can or should be used in teaching and how their usage should be planned by the instructor.

7. CONCLUSIONS

This paper presented a categorization that divides the visualizations according to the level of cognitive development as defined in Bloom's taxonomy. To assure all the different visualization categories are actually achievable, the paper also presented ideas on how to build visualizations on each level.

It is also possible to make exercises to support all the levels of Bloom's taxonomy in a normal way using a pen and a paper, too. The benefit in using visualizations in all the categories was to support the students who have difficulties in perceiving how the program works.

When producing new visualization materials, more attention should be paid on the pedagogical level of the visualizations. When designing the use of visualizations in the course level, the instructor should also take care that all the different categories of the usage of visualizations are covered.

8. REFERENCES

[1] Codewitz, international project for better programming skills. *http://www.codewitz.net*, September 2005.

[2] Macromedia Flash. *http://www.macromedia.com/flash/*, September 2005.

[3] B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. Krahwohl. *Taxonomy of Educational Objectives, Handbook I: Cognitive Domain.* David McKay Company, Inc., New York, 1956.

[4] J. Carter, J. English, K. Ala-Mutka, M. Dick, W. Fone, U. Fuller, and J. Sheard. How shall we assess this? *ACM SIGCSE Bulletin , Working group reports from ITiCSE on Innovation and technology in computer science education*, 35(4):107–123, 2003.

[5] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *ITiCSE 2005, Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 14–18, June 2005.

[6] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, 2004.

[7] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. *Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004*, May 2004.

[8] T. Naps, G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

[9] T. L. Naps, J. R. Eagan, and L. L. Norton. Jhav - an environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE Bulletin , Proceedings of the thirty-first SIGCSE technical symposium on Computer science education SIGCSE '00*, 32(1):109–113, 2000.

[10] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[11] E. Soloway and J. Spohrer. *Studying the Novice Programmer.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989.

[12] A. T. Virtanen, E. Lahtinen, and H.-M. Järvinen. VIP, a visual interpreter for learning introductory programming with C++. *Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education*, pages 129–134, November 2005.

[13] L. E. Winslow. Programming pedagogy – a psychological overview. *SIGCSE Bulletin*, 28(3), September 1996.

Publication (iii)

Ahoniemi, T., Lahtinen, E. Visualizations in Preparing for Programming Exercise Sessions *In: The Proceedings of The Fourth Program Visualization Workshop*, June 2006, Florence.

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 178 (2007) 137–144

www.elsevier.com/locate/entcs

Visualizations in Preparing for Programming Exercise Sessions

Tuukka Ahoniemi[1] Essi Lahtinen[2]

Institution of Software Systems
Tampere University Of Technology
Tampere, Finland

Abstract

Visualizations are widely researched and used in teaching but the results of their benefits in learning are vague. We introduce an experiment of using visualizations in learning introductory programming. The aim was to support students in their preparation for the exercise sessions by using visualizations. The students' preparation consists of two phases that both are supported: reviewing the subject and a homework assignment. Thus this is also a novel approach to using programming visualizations and integrating them to the course content.
The experiment shows positive results especially among the students with no prior programming experience and the students who consider the programming course challenging. We conclude that integrating the use of visualizations to students' preparation for exercise sessions leads to better learning, more meaningful studying, and ultimately to better preparation. Therefore we also suggest this as a possible way for integrating visualizations to the course.

Keywords: Computer Science Education, Programming, Visualizations, Novice Programmers

1 Introduction

The learning problems in programming are often connected to more advanced issues than individual concepts, so the learning materials and situations should also be directed to develop more advanced programming skills [5]. One of the biggest learning problems of the novice programmers is that they have to handle abstract concepts of which they do not have a concrete model in their everyday life [7]. Thus, providing interactive visualizations as extra material for the students is a good way to concretize the subject in the beginning.

The most common use of visualizations is demonstrating a code example as an illustrative visualization. We wanted to make students participate in the visualization and integrate the use of visualizations to the students' preparation and homework

[1] Email: tuukka.ahoniemi@tut.fi
[2] Email: essi.lahtinen@tut.fi

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2007.01.043

assignments for their weekly exercise sessions. The effects of this approach were tested in a real learning situation by in-class tests.

2 Background

The research done on the field of visualizations has resulted in instructions on how to build visualizations so that they will be pedagogically as beneficial as possible. For instance, Naps et al. recommend that the visualizations should engage the student to participate in the visualization actively [6]. As possible ways to do this it is suggested, e.g., that the visualizations should enable the user to provide his own input for the program and that there should be an interactive prediction in the visualization tool [8]. To increase the interactivity of the visualizations they can also be built to support all six stages of cognitive development listed in Bloom's taxonomy [4].

Despite all these recommendations and ideas on how to improve visualizations, the reports on their usage are diverse. A wide study conducted by Hundhausen et al. states that it is more important how the visualizations are used than what their content is [3]. In an other publication, Hundhausen reports that visualizations can actually distract the students' attention away from the subject [2]. On the other hand according to Ben-Bassat Levy et al. visualizations benefit the students with learning problems. This was also our main interest of research [1].

3 The Experiment

This experiment took place on an introductory course for programming *(CS1)* in Tampere University of Technology. The prerequisites for the course are limited to only basic knowledge of computer literacy and it is the first programming course for the students. The programming language used on the course is C++. There are weekly lectures and exercise sessions. The students ought to complete a small homework assignment prior the exercise session. The homework assignment requires them to familiarize themselves with the basics of the new subject. This usually also means reviewing the content of the lectures with the course material.

The idea of visualizations was familiar to the students already before the experiment. We had supported the students' own studying by providing visualizations on the course web page. The printed course material contains web addresses of the visualization examples and the visualization tool – VIP [9] – was also demonstrated on a lecture.

The experiment took place on the fourth and the fifth week of the course. On the first week of the experiment (the fourth week of the course) the exercise sessions dealt with loop structures and on the second week arrays. These weeks were chosen because both of the subjects are typically difficult for novice students [5] and they are easy to visualize.

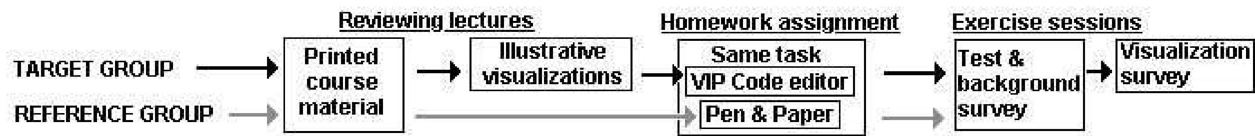


Fig. 1. Organization of the experiment.

3.1 The Method

We used two random groups of about 30 students who had enrolled for the excersise sessions. The target group used visualizations when preparing for the exercise session and the reference group did not. The organization of the experiment is illustrated in Figure 1.

3.1.1 Settings before the Exercise Session

Both groups had the printed course material for reviewing before the exercise sessions. Besides the printed course material, the students in the target group were provided an extra web page with instructions on how to review with the visualization examples and links to the examples. On both weeks the reviewing material contained two *illustrative visualizations* [4] to clarify the concepts.

The actual homework assignments were exactly the same for both groups. The only difference was that the students worked on them using different tools. The reference group had the assignment available on the course web site. Most students in the reference group had used pen and paper to write the code and the answers to the questions. Some of them had also used a regular code editor and a compiler.

The web page provided for the target group contained the homework assignment as text just like for the other students. In addition, there was a link to a visualization tool where the student could start working on the code. VIP [9] contains a code editor where the student can write his own solutions, compile them and run them as a visualization.

3.1.2 Settings in the Exercise Session

On the experiment weeks, there was a short written test in the beginning of the exercise sessions to measure the students' learning. The students were not notified about the test in advance. They were not allowed to look at the materials and they returned their answers anonymously. The task was to write really small programs similar to the ones they had implemented in their homework assignments. The time was limited to only five minutes because the tasks tested the very basics and therefore would have been easily implemented in the time – assuming the subject was well learnt. We also wanted to have more variation inside the groups by limiting the time. Only the best students would complete the whole test.

Besides the small test, all the students responded to a short survey for background information, e.g., about their previous programming experience and how they felt about their progress on the course. Also the amount of time used, both on reviewing the subject and on doing the actual homework assignment, was asked. The students in the target group also answered another survey concerning the use

	Target group	Reference group
novices & strugglers	12	17
others	9	10

Fig. 2. The focused subset (highlighted with grey) was the novices and the strugglers of both groups. The amounts of students are shown in the table.

of the visualizations as a supporting tool for exercise preparation. The survey form was attached to the test so that the background information can be connected to the test answers.

3.2 The Homework Assignment

The exercise sessions in the first week dealt with loop structures. In the homework assignment there was a simple example of a `while`-loop. The task was first to find out what the piece of code does and to understand how it works. Then the students had to modify the code to implement an other kind of a functionality. The task reaches the level application (3) of Bloom's taxonomy of cognitive development, since it requires ability to apply one's knowledge in a new situation. Thus the assignment version implemented in the visualization tool is a *utilizing visualization* [4].

The subject in the second week was arrays. To widen the perspective of visualizational aid we chose this homework assignment differently: The students familiarized themselves with a given complex loop structure handling two arrays and answered questions related to it. The task requires identifying and analyzing the components of the code, so it is on the level analysis (4) of Bloom's taxonomy. Thus the version implemented in VIP is an *analyzable visualization* [4].

4 Results

On the first week, there were 21 students present in the exercise session of the target group and 27 students in the reference group, i.e. alltogether 48 students. On the next week the corresponding numbers are 21, 22 and 43.

As visualizations are mainly targeted for the novices and the students who have learning difficulties, we constricted the comparison of the groups to only the novices (no previous programming experience) or the ones finding the course subjects so far difficult or very difficult (here called *the strugglers*). The division and the numbers of the students in the groups is illustrated in Figure 2.

The results are divided into two parts: the effects on learning results and the effects on studying behaviour. The first represents the students' knowledge on the subject measured in the test as the second represents how the students prepared for the exercise session.

According to the results from the first week, the use of visualizations benefits learning: we found a statistically significant difference of the mean values of the test grades getween the groups. The results from the second week are analogous and support the results from the first week. Because of the smaller difference in the second week, this section mainly concentrates on representing the results from the

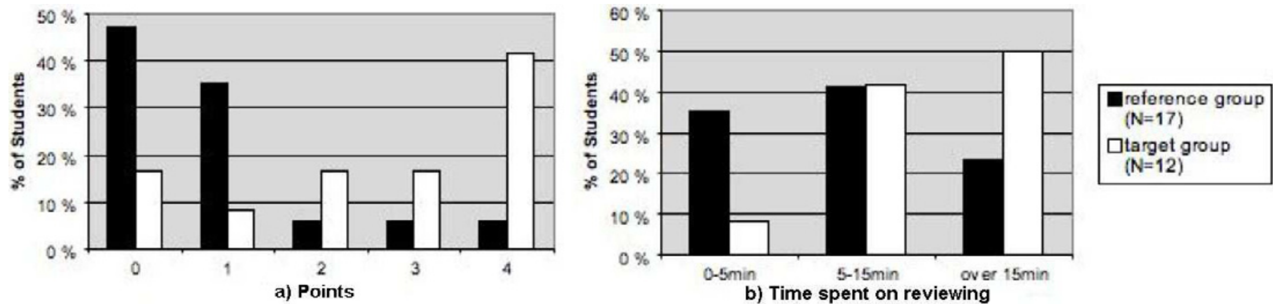


Fig. 3. Results from the first week of the experiment concerning the novices and struglers: (a) Distribution of the grades of the first task of the test and (b) Time spent on reviewing.

first week.

4.1 The Effects on Learning Results

The effects on learning results were analyzed by rating the students' answers for the test. For example, on the first week all three tasks were graded on a linear scale with points from 0 to 4 resulting the maximum of 12 points. On the second week the maximum was only 8.

The loop tasks seemed to be difficult for the students to complete in the given five minutes. The mean result was altogether only 3.5 out of 12 points (standard deviation 2.5). An independent samples T-test was used to analyze the difference between the groups. The means for the focused subset of novices and strugglers are 3.6 points (standard deviation 2.2) for the target group and only 1.7 points (standard deviation 1.5) for the reference group. This shows a significant statistical difference ($p < 0.05$). Even if the comparison is done to the whole groups (instead of only the focused subset) there is a small analogous difference between the groups.

In the next week, the corresponding means of the novices and strugglers are 3.1 points out of 8 points (standard deviation 2.3) for the students in the target group and 2.3 points (standard deviation 1.9) for the ones in the reference group. The trend is same as on the earlier week.

As the students carried out the tasks in the test sequentially, they all started with the first task. Figure 3a shows the percentage values of each grade in this task. Only the novices and the strugglers are taken into account. Almost all students in the target group (10 out of 12 = 83%) got at least one point and even 42% full 4 points as the reference group had the same numbers in 53% and 6%. The same phenomenon can be observed in the results of the second week.

4.2 The Effects on Studying Behaviour

Since the novices and the strugglers were the only ones whose learning results are different, it is logical that they are the only ones' whose studying behaviour was influenced by the visualizations. Thus this subsection concentrates only on the novices and strugglers of the groups.

According to the students' answers to the survey about their preparation, the students in the target group had used more time than the students in the reference group. Both the time spent on reviewing the subject and the time spent on doing

the homework assignment were higher. The difference was bigger in reviewing the subject. The comparison between the time usage on reviewing the subject in the first week of the experiment is shown in Figure 3b.

More than a third of the students in the reference group spent less than 5 minutes in reviewing. More than 90% of the students in the target group spent longer than 5 minutes. It is clear that the students using the visualization tool concentrated longer even though the statistical significance between the groups can not be stated.

Also the feedback of the survey about visualizations as a preparation tool resulted in plain positive feedback. Students wrote comments like "Though having read the specified course material, I really understood the subject after using the visualization examples."

4.3 Comparing the Results of the Two Weeks

The experiment was not done in a strictly controlled situation but in a normal teaching group so some circumstances varied between the two weeks of the experiment. E.g., there were more absent students on the second week. The subjects on the two weeks were different so we also had a new type of homework assignment and a different test on the second week. All of these factors have influenced the results.

On the first week, the homework assignment was a *utilizing visualization* and on the second week an *analyzable visualization*. One important reason for the difference in the results can be that *utilizing visualizations* engage the student to produce his own code where as *analyzable visualizations* engage the student to observe the code intensively. The test performed in the class room was about producing their own code. So on the first week the preparation and the test were more similar than on the second week.

The in-class test was not announced in advance so on the first week of the experiment no one expected it. On the second week the students might have assumed that there could be a test again. Thus the students may have prepared better for the exercise session. This can also be one of the reasons why the statistical difference was not achieved on the second week.

5 Discussion

Even if the circumstances between the weeks of the experiment varied, it is advantageous that the experiment was done in a real learning situation. We captured the students' experiences in a situation where they act as they would act normally when studying. Thus the results can better be applied to planning teaching in the future.

The results show that the use of visualizations helped the students who have most challenges in learning programming (the novices and the strugglers). They learnt more if they used visualizations when preparing for the exercise sessions. The students who had earlier experience in programming already had a mental model about the subject and thus the use of visualizations was not so helpful. Also the

students who felt that the subject was easy could form the mental model without using visual materials. Hence, they did not benefit of the use of visualizations so much either.

Another result was that the students who used visualization examples along with the normal course material spent more time on reviewing the subject than the others. Studying obviously became more interesting as a new visual perspective was provided.

So what really can be concluded from the results is that visualizations do aid learning, but it is not sure whether this results directly of their usage. It can also result from the fact that when using visualizations, the studying itself is more interesting and the students use more time on it and thus learn better. However, it is not important, if the visualizations improve the learning results directly. The most important result is that they do improve them.

The difference between the two weeks of the experiment – the week when the students did a *utilizing visualization* exercise and the week when they did an *analysable visualization* exercise – also supports the recommendation from Naps et al. that the visualization should engage the student to work actively [6]. *Utilizing visualization* makes the student produce their own code where as *analysable visualization* only makes them analyze code written by someone else. The engagement to the visualization is more intense with a *utilizing visualization*. Also the learning results from the week when the *utilizing visualization* was used are better.

Using visualizations in students' preparation for exercise sessions had definitely a positive outcome because of the better learning. The exercise sessions ran smoother because students were better prepared due to the increase in their motivation. This also shows that using visualizations in preparing for exercise sessions is a working way of integrating visualizations to the rest of the course content.

The problems and considerations of this kind of approach are technical issues and the time spent by the teacher. Implementing tasks with visualizations requires quite advanced tools that have to be available for every student. Also preparing the tasks with a visualization tool takes more effort from the teacher than without a visualization tool.

When planning new ways to use visualizations in a course the teacher should also bear in mind that not all want to use new kinds of learning tools. As the use of visualizations mainly benefit the novices and the strugglers, it can be annoying for the students that do not need it. Some of the students might not like visual learning style or just have their own idea on how to work. Thus we recommend that the use of visualization tools is optional.

6 Conclusions

Using program visualizations improve the learning of students with no earlier programming experience and the students who have difficulties in programming. We cannot say whether the better learning results originate from the pedagogical impact of the visualizations or from the fact that the visualizations made the students

study for a longer time. Either way, using visualizations improved the students' learning and preparation for the exercise sessions which was the purpose. Therefore, we recommend both using visualizations in teaching and using the exercise sessions to integrate the visualizations to the other parts of the course.

References

[1] Ben-Bassat Levy, R., M. Ben-Ari and P. A. Uronen, *The Jeliot 2000 program animation system*, Computers & Education **40** (2003), pp. 1–15.

[2] Hundhausen, C. D., *Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach*, Computers & Education **39** (2002), pp. 237–260.

[3] Hundhausen, C. D., S. A. Douglas and J. T. Stasko, *A meta-study of algorithm visualization effectiveness.*, Journal of Visual Languages & Computing **13** (2002), pp. 259–290.

[4] Lahtinen, E. and T. Ahoniemi, *Visualizations to Support Programming on Different Levels of Cognitive Development*, Proceedings of The Fifth Koli Calling Conference on Computer Science Education (2005), pp. 87–94.

[5] Lahtinen, E., K. Ala-Mutka and H.-M. Järvinen, *A study of the difficulties of novice programmers*, ITiCSE 2005, Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (2005), pp. 14–18.

[6] Naps, T., G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. Velazquez-Iturbide, *Exploring the role of visualization and engagement in computer science education*, SIGCSE Bulletin **35** (2003), pp. 131–152.

[7] Robins, A., J. Rountree and N. Rountree, *Learning and teaching programming: A review and discussion*, Computer Science Education **13** (2003), pp. 137–172.

[8] Rössling, G. and T. L. Naps, *A Testbed for Pedagogical Requirements in Algorithm Visualizations*, ITiCSE 2002, Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (2002).

[9] Virtanen, A. T., E. Lahtinen and H.-M. Järvinen, *VIP, a visual interpreter for learning introductory programming with C++*, Proceedings of The Fifth Koli Calling Conference on Computer Science Education (2005), pp. 125–130.

Publication (iv)

Lahtinen, E., Ahoniemi, T. Kick-Start Activation to Novice Programmers - A Visualization-Based Approach *In: The Proceedings of PVW 2008 - Fifth Program Visualization Workshop*, July 2008 Madrid, Spain.

ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 224 (2009) 125–132

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Kick-Start Activation to Novice Programming — A Visualization-Based Approach

Essi Lahtinen[1] and Tuukka Ahoniemi[2]

Department of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

In the beginning of learning programming students have misconceptions of what programming is. We have used a kick-start activation in the beginning of an introductory programming course (CS1) to set the record straight. A kick-start activation means introducing the deep structure of programming before the surface structure by making the students solve a certain type of problem in the first lecture. The problem is related to a realistic computer program, simple enough for everyone to understand and allow students to participate in debugging. A visualization-based approach helps making the example more concrete for students.
In this article we present the concept kick-start activation and one concrete example. To support the example, we have also developed a visualization using the visualization tool JHAVÉ. We got positive feedback on the example and suggest further development of kick-start activations in order to make the beginning of learning programming more motivating for students.

Keywords: Teaching programming, Novice programmers, Visualizations, Kick-start activation.

1 Introduction

Students who enroll to introductory programming courses (CS1) have plenty of misconceptions about the nature of programming and some students do not know what programming is at all. The course typically starts with the teacher trying to correct the misconceptions by emphasizing that programming is more problem-solving and thinking than typing program code. The concept of an algorithm is introduced, as well as some tools for implementing algorithms and designing programs, such as pseudocode or flow charts.

A classical first example of an algorithm is a recipe in a cook-book. A recipe is a relatively unambiguous, detailed set of instructions. If you follow the instructions carefully you will have a food portion as the result. However, there are problems

[1] Email: essi.lahtinen@tut.fi
[2] Email: tuukka.ahoniemi@tut.fi

1571-0661/© 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.12.056

with this example. Firstly, it is not at all related to computers. Thus students might feel that the teacher is stating the obvious or even explaining nonsense when he/she is talking about cooking and algorithms instead of programming. Secondly, even though comparing cooking recipes and algorithms gives a clear idea on what an algorithm is, it does not really help to understand what a programmer does. There is no occupation where the job is to develop new recipies. Thus, the underlying idea of programming is not delivered to the students. Thirdly, the methaphor also does not help in explaining the programming process for the student. There are no concrete examples on important phases like designing, testing nor debugging. The student might still continue carrying the misconception of programming being merely the implementation of an algorithm.

We introduce a different way to start the course: *kick-start activation*. In this approach, we get into the deep structure of programming before the surface structure is even introduced. Our target audience is especially the students who do not know anything about programming before the kick-start activation.

In this article we first present the idea of a kick-start activation in Section 2. Then we introduce our example and explain how we use it in Section 3. Section 4 presents the visualization and feedback. Finally, discussion and conclusion are included in Section 5.

2 Criteria for a Kick-Start Activation

In our opinion, to make the opening of the course interesting for students, one needs to get directly into the real problems, i.e., a problem that requires an algorithmic solution. In the case of programming this means skipping the surface structure, such as the syntax of the programming language, and starting from the deep structure of programming, i.e., a problem that the students solve themself. We call this kind of an introduction *kick-start activation* because it is a fast-forward jump-in approach and it engages students in the example since they solve the problem.

Our *main criterion* for the example presented in the kick-start activation is that it has to be based on *a real computer program*. The benefits of a real programming example are:

- In addition to introducing the concepts of algorithms, pseudo code and flow charts one can also introduce
  - · problem solving and the phases of programming,
  - · the idea of testing algorithms and the idea of testing programs, and
  - · what the work of a programmer is like.
- It helps to explain the difference of human thinking and the way the computer works.
- The execution of the algoritm can be explained and demonstrated with a computer.
- One can also show a an implementation in a programming language to give an example. Students can identify the control structures of the pseudo code from the program code. Even if the students do not understand theprogramming language

syntax yet, it gives them a concrete example on what a programming language looks like.

- It can be concretized by a program visualization that the students can run.

Our *second criterion* is that the kick-start activation needs to be simple enough so it can be understood by everyone. We decided that it has to be an example that *relates to everyday life*. Besides that we chose not to use a real programming language nor any terms, pictures, or other details that relate to computers. For example, we did not want `[]`-operators in the algorithm or memory addresses in the pictures. These would just add extra details that are irrelevant at this stage. Instead of using a programming language it is easier to fade out the surface structure of programming by using a natural-language-like pseudo code presentation and flow charts. To concretize the pseudo code and flow chart we developed a visualization that illustrates how the algorithm would be run by a computer if the computer could understand it.

The *third criterion* for a kick-start activation was to make *students take part* in the example. As programming is much more thinking and problem-solving than using the programming language syntax, there are numerous programming related activities that students can try already in the beginning of the course. For instance testing an algorithm is a task that can be given to a student. One practical way of doing this is developing a buggy version of an algorithm that the students can debug.

3 Our Example: Hyphenating Finnish Words

The topic of our kick-start activation was the hyphenation rules of the Finnish language. Word processors have spell checking and automatic hyphenation, i.e., computer programs are hyphenating Finnish words. In addition, every student knows how to spell [3] so the topic is general enough.

The exact rules for hyphenating Finnish are not common knowledge in Finland even if it is easy to hyphenate Finnish for everyone who knows how to speak the language. Fortunately the rules are simple enough to be explained to students in a few sentences. Still, it is non-trivial to build a hyphenation algorithm. The algorithm requires a loop structure to go through the letters of the hyphenated word and a couple of if-statements to choose which hyphenation rule to apply.

For example, the first of three hyphenation rules called *the consonant rule* states the following: *if there is a vowel followed by one or more consonants, a hyphen is placed directly before the last consonant.* The window on the right hand side in Figure 1 presents the algorithm based on the rules. The consonant rule can be identified in the marked area of the figure.

A word is a data structure that can be understood even without knowing the data type `string`. A word can also be drawn like a line of alphabet building blocks (See the window on the left hand side in Figure 1). Introducing the computer

[3] In this situation actually: *in Finland* every student knows how to spell *Finnish*.

memory or other similar details for the student is unnecessary. Drawing the data structure as a line of building blocks actually allows us to visualize the addition of a hyphen: a picture animation where a block with the character '-' slides and slips in between the blocks of the word.

On the lecture, our intention was to highlight that designing and testing the algorithm with pen and paper is a big part of programming. To describe this clearly we used a three step example:

(i) First we quickly designed a hyphenation algorithm. Though it seemed to be correct the hasty design had on purpose produced a buggy solution.

(ii) Then the students tested the algorithm and hopefully found the error. After this we discussed how important it is to understand the problem before you start designing the algorithm.

(iii) Finally, we explained the hyphenation rules deeper for the students and designed a new algorithm properly. The final result was a correctly working algorithm.

The example included two algorithms. We call these *the premature algorithm* (produced in step 1) and *the mature algorithm* (produced in step 3).

The purpose of the testing phase was to activate the students. They were actually performing a programming related task even if they thought they did not know any programming yet. The idea is that the students can use the visualization to run and test the algorithm. The testing could of course be done using only pen and paper, but the visualization is handy in it. We gave a link to the visualization to the students for later use so that they could revise the lecture using the visualization.

4 The Visualization

There are many program visualization tools available for presenting basic programming structures for novice programmers for instance, Jeliot 3 [4] for Java, VIP [12] for C++, and Ville [7] and Planani [9] for multiple different languages. These visualization tools work on program code level, so they assume that the student already understands some programming language and thus are not suitable for our target audience. There is also a visualization tool called RAPTOR [1] where the students can construct flow charts and the tool will visualize them for the student. The RAPTOR flow charts are also close to the program code level, e.g., the tool shows the content of variables and arrays.

We needed a completely syntax-free common purpose visualization tool where we can write the algorithm in a few Finnish sentences and draw the building blocks exactly according to our needs. Thus, the existing program visualization tools did not suit our purposes. However, in the field of algorithm visualizations there was one tool flexible enough: JHAVÉ [6] and its Gaigs support class package. With a bit of imagination we were able to use this algorithm visualization tool slightly unorthodoxically and produce the hyphenation visualization.

The info screen of JHAVÉ's execution window is normally used for showing

algorithm specific instructions written in HTML. The tool allows the use of images as a part of the HTML page with the <image> tag. This feature let us implement the flow chart animation with a set of fixed images. The images were then presented in the correct order by showing a particular image in each state of the program. With the possibility of using HTML and images in JHAVÉ, one could design many sorts of examples as the technical implementation is limited solely to the creation of the images.

Using JHAVÉ, we implemented two different presentations of *the hyphenation algorithm visualization*: a pseudo code view and a flow chart. Both of these presentations also contain a window with the alphabet building block picture of the hyphenated word. Screenshots can be seen in Figure 1 and Figure 2. There were two different algorithms that we visualized: the premature and mature. Since there are two different presentations of both the algorithms we actually had four different visualizations.

The student can control the visualization using the step and step-back buttons. The execution of the algorithm is visualized by coloring the nodes in the flow chart or the lines of the pseudo code synchronously with the steps. As the program is hyphenating words, the state of the word in each step is visualized in the window with the alphabet building block picture on the left hand side. There are pictures of two words: the original word without the hyphens and the result where the hyphens are added as the algorithm proceeds. The visualization also colors the alphabet building blocks that the algorithm is handling.

4.1 Student Engagement

According to research on the field of visualizations, student engagement is vital for learning when a student uses visualization [11]. Naps et al. [5] present a Visualization Engagement Taxonomy that describes six levels of learner engagement with visualization technology. On top of the lowest level of existing engagement—*Viewing*—are the more active levels: *Responding* and *Changing* an existing visualization and *Constructing* and *Presenting* ones own visualization.

As the algorithm is given fixed in the hyphenation algorithm visualization, the student engagement is enhanced by allowing the student to provide his/her own input word for the algorithm. This corresponds to the level *Change* of the Visualization Engagement Taxonomy [5]. To attain the level *Response* also, the flow of the program is interrupted with pop-up questions querying about the next behavior of the program.

4.2 Student Feedback

We evaluated the visualization with a quantitative survey after the lecture where we used it. We handed in a questionnaire on paper for the students. We received altogether 113 responses. 71 of the respondents (63%) had no programming experience before the course.

The feedback was generally positive since 53% of the respondents said that the

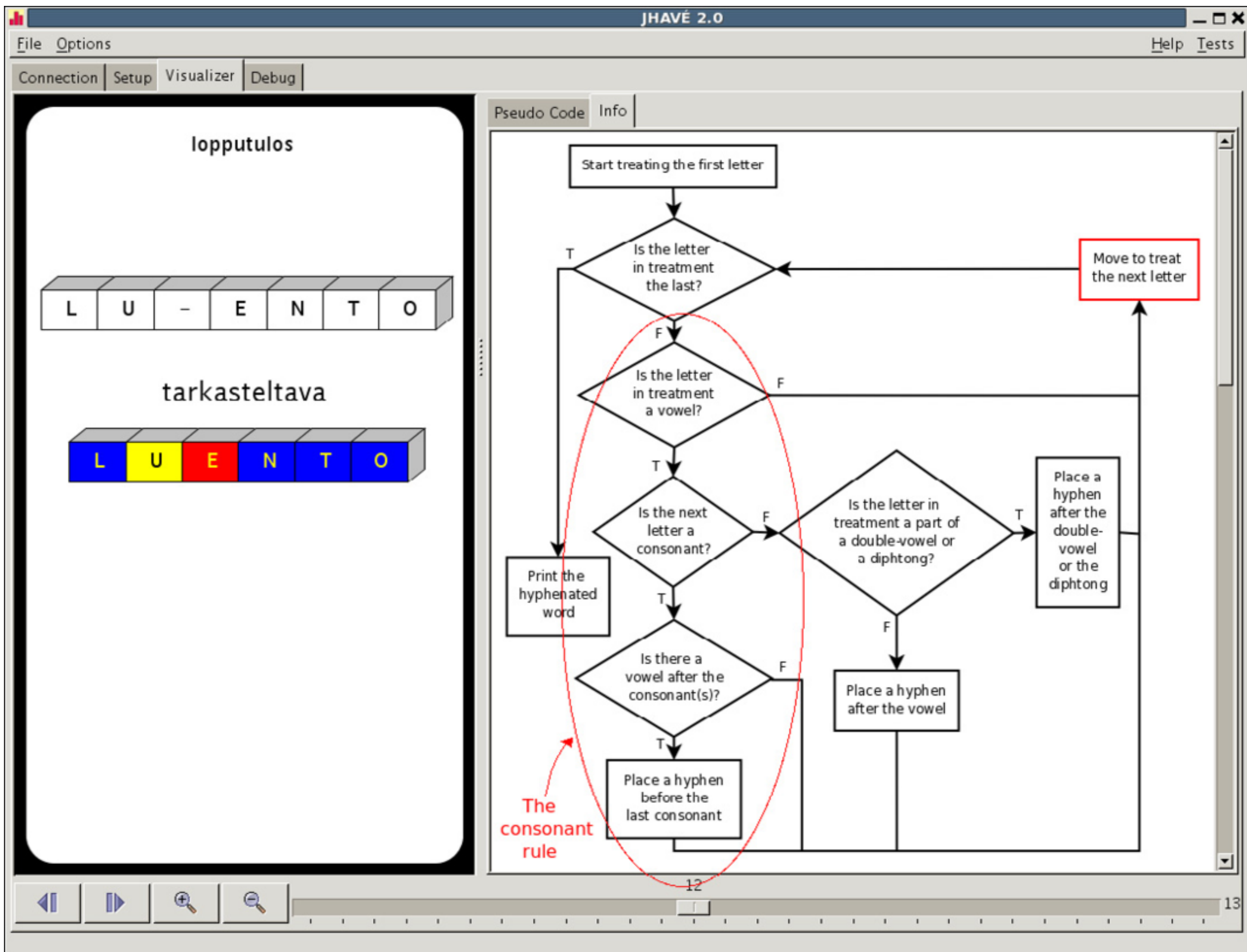


Fig. 1. The flow chart version of the visualization.

```
JHAVÉ 2.0
File  Options                                            Help  Tests
Connection | Setup | Visualizer | D

Question
Mille riville mennään seuraavaksi?
○ 8
○ 14
○ 4
○ 13
Check Answer

lopputulos
L U - E N T O
tarkasteltava
L U E N T O

Pseudo Code | Info
Tavutusalgoritmi

Tavuta sana
 1: Ota käsiteltäväksi sanan ensimmäinen kirjain
 2: WHILE ei olla vielä viimeisessä kirjaimessa DO
 3:     IF käsiteltävänä on vokaali THEN
 4:         IF käsiteltävästä seuraava kirjain on konsonantti THEN
 5:             IF viimeisen peräkkäin olevan konsonantin jälkeen on vokaali THEN
 6:                 Sijoita tavuviiva ennen viimeistä konsonanttia
 7:         ELSE
 8:             IF käsiteltävä vokaali on tuplavokaalin tai diftongin alku THEN
 9:                 IF tuplavokaalin tai diftongin jälkeen on vielä vokaali THEN
10:                     Sijoita tavuviiva tuplavokaalin tai diftongin perään
11:             ELSE
12:                 Sijoita tavuviiva käsiteltävän vokaalin perään
13:     Siirry käsittelemään seuraavaa kirjainta
14: Tulosta koko sana tavuviivoineen

14
Started Logging Connection
```

Fig. 2. The pseudo code version of the visualization with an activating pop-up question.

visualization looked nice (agree or totally agree), 86% thought that is was useful for learning (agree or totally agree), and only 5% thought that it disturbed the lecture (agree or totally agree).

We performed a cross tabulation and a χ^2-test for some of the variables and found out that the students with no earlier programming experience thought that the vi-

sualization was more useful for learning than the students who had programmed before coming to the course. This difference is statistically significant ($p < 0,05$). The reason is also obvious: the students with earlier programming experience already had an understanding on how algorithms and flow charts work so they do not need the visualization for understanding the hyphenation algorithm. This result shows that we managed to help the students who were the target audience of the visualization.

After all, the most important feedback was that our students were listening to the hyphenation example intensively on the lecture. Two teachers tried the example and both of them could sense a notable difference in the lecture situation compared to the cook-book example.

5 Discussion and Conclusions

The kick-start activation received positive feedback both from the students and the teachers who used it. We think that our approach was successful because the criteria were designed carefully and there was a visualization tool that aided both presenting the example and understanding it. This example could be used as a source of ideas for other topics to build kick-start activations of.

There are not many program visualization tools available for our target audience—the students who do not know anything about programming yet. In addition to our visualization we have found a system called SICAS [3] that could probably also be used for presenting a kick-start activation. It is based on similar principles and allows students to construct their own flow charts and visualize them. However, currently it is not used the same way we used our visualization.

The conceptual framework of programming knowledge developed by McGill and Volet [2] suggests that in addition to syntactic and conceptual knowledge a programmer also needs strategic knowledge of programming. Reports on the state of field show that visualizations are often used for only presenting programming concepts [10]. The scope of our visualization is more in the strategic knowledge since it focuses on the programming phases: testing and design.

In the development of the visualization we also emphasized student engagement in the levels of the Visualization Engagement Taxonomy [5]. The visualization is most activating when the student is guided to use it in the three step lesson we described in Section 3. This requires either a teacher to explain the hyphenation problem and the need for debugging the first version of the algorithm or the student to read this from the material by himself. The idea of connecting a visualization to a certain study material is similar to the one presented in an ITiCSE working group report about hypertextbooks [8]. We think that the visualization of the mature version of the algorithm could also be used without the debugging phase just for presenting the concepts algorithm, pseudo code, and flow chart. This way the example would be less challenging and the activation of the student would be left only to the pop-up questions.

The best possibility for activating students would be to make them correct the

bug or build a completely new correct algorithm after finding the bug from the premature version of the algorithm. This can, however, be very challenging for a novice student so we did not try it. It would be an interesting future work idea to build a visualization tool where the student could build the correct algorithm by modifying the flow chart. Another idea for future work is that we could implement different kinds of premature algorithms. There could be easier and more difficult bugs for the debugging task.

6 Acknowledgments

Special thanks to Prof. Thomas Naps (University of Wisconsin, Oshkosh) for his enormously useful guidance with visualizations in common and help with the JHAVÉ visualization tool. Nokia Foundation has partly funded this work.

References

[1] Giordano, J. C. and M. Carlisle, *Toward a more effective visualization tool to teach novice programmers*, in: *SIGITE '06: Proceedings of the 7th conference on Information technology education* (2006), pp. 115–122.

[2] McGill, T. and S. Volet, *A conceptual framework for analyzing students' knowledge of programming*, Journal on research on Computing in Education **29** (1997), pp. 276–297.

[3] Mendes, A. J., A. Gomes, M. Esteves, M. J. Marcelino, C. Bravo and M. A. Redondo, *Using simulation and collaboration in cs1 and cs2*, SIGCSE Bull. **37** (2005), pp. 193–197.

[4] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with Jeliot 3*, Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004 (2004).

[5] Naps, T., G. Rössling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. Velazquez-Iturbide, *Exploring the role of visualization and engagement in computer science education*, SIGCSE Bulletin **35** (2003), pp. 131–152.

[6] Naps, T. L., J. R. Eagan and L. L. Norton, *JHAVé - An environment to actively engage students in web-based algorithm visualizations*, ACM SIGCSE Bulletin , Proceedings of the thirty-first SIGCSE technical symposium on Computer science education SIGCSE '00 **32** (2000), pp. 109–113.

[7] Rajala, T., M.-J. Laakso, E. Kaila and T. Salakoski, *VILLE – A language-independent program visualization tool*, in: *Proceedings of The Seventh Koli Calling Conference on Computer Science Education*, 2007.

[8] Rössling, G., T. Naps, M. S. Hall, V. Karavirta, A. Kerren, C. Leska, A. Moreno, R. Oechsle, S. H. Rodger, J. Urquiza-Fuentes and J. Ángel Velázquez-Iturbide, *Merging interactive visualizations with hypertextbooks and course management*, in: *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education* (2006), pp. 166–181.

[9] Sajaniemi, J. and M. Kuittinen, *Visualizing roles of variables in program animation*, Information Visualization **3** (2004), pp. 137–153.

[10] Shaffer, C. A., M. Cooper and S. H. Edwards, *Algorithm visualization: a report on the state of the field*, in: *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education* (2007), pp. 150–154.

[11] Stasko, J. T. and C. D. Hundhausen, *Algorithm Visualization*, in: *Computer Science Education Research* (2004), pp. 199–228.

[12] Virtanen, A. T., E. Lahtinen and H.-M. Järvinen, *VIP, a visual interpreter for learning introductory programming with C++*, Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education (2005), pp. 129–134.

Publication (v)

Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O. Review of Recent Systems for Automatic Assessment of Programming Assignments *In: The Proceedings of 10th Koli Calling International Conference on Computing Education Research*, October 2010, Koli, Finland.

Review of Recent Systems for Automatic Assessment of Programming Assignments

Petri Ihantola
Department of Computer Science and Engineering
Aalto University
petri@cs.hut.fi

Tuukka Ahoniemi
Digia Plc
Finland
tuukka.ahoniemi@digia.com

Ville Karavirta
Department of Computer Science and Engineering
Aalto University
vkaravir@cs.hut.fi

Otto Seppälä
Department of Computer Science and Engineering
Aalto University
oseppala@cs.hut.fi

ABSTRACT

This paper presents a systematic literature review of the recent (2006–2010) development of automatic assessment tools for programming exercises. We discuss the major features that the tools support and the different approaches they are using both from the pedagogical and the technical point of view. Examples of these features are ways for the teacher to define tests, resubmission policies, security issues, and so forth. We have also identified a list of novel features, like assessing web software, that are likely to get more research attention in the future. As a conclusion, we state that too many new systems are developed, but also acknowledge the current reasons for the phenomenon. As one solution we encourage opening up the existing systems and joining efforts on developing those further. Selected systems from our survey are briefly described in Appendix A.

1. INTRODUCTION

Assessment provides the teacher with a feedback channel that shows how learning goals are being met. It also ensures for an outside observer that students achieve those learning goals. Assessment provides both means to guide student learning and feedback for both the learner and the teacher about the learning process – from the level of a whole course down to a single student on some specific topic being assessed.

Students often direct their efforts based on what is assessed and how it affects the final course grade [6, Chapter 9]. Continuous assessment during a programming course ensures that students get enough practice as well as get feedback on the quality of their solutions. Providing quality assessment manually for even a small class means that feedback can not be as instant as in one-to-one tutoring. When the class size grows, the amount of assessed work has to be cut down or rationalized in some other way. Automatic assessment (AA), however, allows instant feedback without the need to reduce exercises.

Why do so many automatic assessment systems exist, and why are new ones created every year? Many systems share common features and it would seem that systems already exist that fulfill most assessment needs.

One clear reason for the variety of tools has to do with their availability and lifespan. Tools are often created as a part of a thesis or for a particular course. They are finished enough for studying a research question or to support the needs of one particular course, but are not suitable for distribution. It is rather common that the very first version of a tool was something that the teacher did quickly for his/her very own purpose. These tools might get publicized if some research was the original motivator, but as they never emerge as supported pieces of software, similar systems get implemented again and again. Correspondingly, there are far less systems that are widely adopted than there are papers about new tools.

We argue that presenting *a big picture* about the recently developed and currently available AA systems would help both teachers find the tools they might be searching and developers avoid reinventing the wheel. Literature survey is one way to achieve this. In this survey, our goal is to serve teachers who need to give grades to large classes. This is where the automatic grading of programming assignments can free the teachers' time significantly for doing something else, that can not be automated [9].

Related research, with focus on related surveys, is presented in Section 2. The exact research questions and the methodology used in this survey are described in Section 3. Results are introduced in Section 4. Selection of AA systems, also mentioned in Section 4, are presented in Appendix A. Conclusions, some recommendations based on the data, and our expectations related to the future trends in automatic assessment of programming assignments are discussed in Section 5.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Koli Calling '10, October 28-31, 2010, Koli, Finland
Copyright 2010 ACM 978-1-4503-0520-4/10/10 ...$10.00.

2. RELATED WORK

Tools are an actively researched approach to support teaching programming. For example, a survey by David Valentine found that 18% of the papers published in SIGCSE conference between 1983 and 1993 were tools papers, whereas between 1994 and 2003 the number was 24.6% [78]. In the *Survey of Literature on the Teaching of Introductory Programming* by Pears et al. [55] from 2007, tools were the single largest group among papers classified between tools, curricula, pedagogy, and programming languages. Analysis by Sheard et al. [68] from 2009 also supports the importance of both assessment and tools. Top three themes in their classification of CS education research papers were: 1) ability / aptitude / understanding (40%), 2) teaching / learning / assessment techniques (35%), and 3) teaching / learning / assessment tools (9%).

Pears et al. have also summarized that tools that support teaching programming can be divided into:

- visualization tools (e.g. ANIMAL [60], Jeliot [49], and Tango [73]),
- automated assessment tools (e.g. TRAKLA2 [40], WEB-CAT [16], and BOSS [37]),
- programming support tools (e.g. BlueJ [5]), and
- microworlds (e.g. Karel [54], and Alice [12]).

As stated before, *our focus is on tools for automated assessment of programming assignments.*

There are a few surveys of AA in the context of programming assignments. *A Survey of Automated Assessment Approaches for Programming Assignments* [1] by Kirsti Ala-Mutka from 2005 concentrates on what features of programming assignments are automatically assessed whereas Douce et al. [14] review the history of the field from 1960s to 2005. One of the main findings by Ala-Mutka is that dynamic analysis – that is, assessment based on executing the program – is often used to assess functionality, efficiency, and testing skills. Static checks that analyze the program without executing it are used to provide feedback from style, programming errors, software metrics, and even design. Tools that cover both static and dynamic testing are also well presented in the survey. There are many features to assess, and Ala-Mutka concludes that the selected AA approach should always be pedagogically justified. Although we believe a lot has been done since 2005, a recent survey from 2009 by Liang et al. [42] provides little new to the work of Ala-Mutka.

ITiCSE 2003 working group led by Carter conducted a survey among CS educators (not only programming) to get a snapshot of AA practices and an analysis of respondents' perceptions of automatic assessment [9]. One interesting finding was that the teachers who were not familiar with AA considered its potential more limited than the respondents with experience from AA.

Not all programming exercises can be automatically assessed. Several articles discussed how to design good assignments from the pedagogical standpoint. However, how to deal with the restrictions set by the automatic assessment is not often addressed. Forišek investigated International Olympiads in Informatics[1] (an event similar to the ACM International Collegiate Programming Contests[2]) and found that certain types of assignments they used were unsuitable for automatic assessment [17]. Forišek presents concrete examples of bad assignments (i.e easy to cheat tests) and heuristics on how to detect them. Greening, on the other hand, suggests that programming assignments should be more open in nature instead of satisfying a strict set of specifications often required by automatic assessment [25]. Furthermore, we believe that the very fact that the assessment is automatic is likely to change how some students approach the exercise. Knowingly submitting a weak or even incorrect solution that gets accepted by a machine is quite likely more socially acceptable than trying to cheat a person.

Some of the research outside CS education research will also help us understand when to apply AA and when not to. For example, what kind of problems in code can be detected automatically and what not has been investigated (e.g. [45]).

[1] http://ioinformatics.org/

[2] http://cm.baylor.edu/

3. RESEARCH QUESTION AND METHOD

Based on the previous section, we conclude that the trends and improvements in automatic assessment of programming assignments from the last five years have not been systematically collected. Thus, the following research questions are addressed in this paper:

1. What are the features of automatic assessment systems reported in the literature after 2005[3]?

2. What future directions are indicated?

To answer the questions, a systematic literature review was carried out. This means that an explicit procedure in selecting the systems and papers was applied (see [7] for details): to be included, a paper must have presented an AA system providing *summative*, *numerical* feedback from *programming assignments* or described results from using such system.

By programming assignments we mean assignments where students write code and submit it for assessment. Therefore AA of diagrams (e.g. [76]), AA of algorithm simulation (e.g. [40]), and other visualization based approaches (e.g. providing formative feedback based on visualizations) are not included. In addition, we only included systems where first hand experience was reported. This means that classical systems often mentioned in the related work section (e.g. CourseMarker [30], Assyst [34], etc.) are left out – unless experiences from those systems were reported in the literature we surveyed.

We collected the data by searching for phrases ('automatic' OR 'automated) AND ('assessment' OR 'grading') AND 'programming' from the conference proceedings and journals through *ACM Digital Library* and *IEEE Xplore*. We then applied the inclusion criteria to the abstracts and finally read all the remaining papers. Search terms were collectively decided after first manually examining (reading the abstract and scanning the rest) all the articles of ITiCSE proceedings and three journals: Computer Science Education, Olympiads in Informatics, and Transactions on Computing Education (formerly Journal on Educational Resources in Computing) between 2006 and 2010.

Because of our inclusion criteria, not all of the systems included in this survey are first published after 2005. A

[3] 2005 is when the survey of Ala-Mutka was carried out.

system might have been published earlier but an evaluation study or something similar after 2005.

We applied an iterative process to find the consensus about how to group features of the systems (i.e. subsections of Section 4). We read a selection of papers, made the first version of categories, read more papers and revised the categories. This was repeated until no significant features leading into new categories were found. Some of our results are explained by our background in automated assessment. As a short summary, our attitudes towards automation are positive, we have all used AA several years in our courses, and we have been developing AA and other educational systems.

4. RESULTS

In this section, we introduce the features identified in the literature survey. Systems are cited, but the focus is on features, not on systems.

4.1 Programming Languages

A majority of the systems are either targeted only for Java or have support for 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 include C/C++, Python, and Pascal. Pascal was especially used in most of the competition platforms, where a typical language support also included C/C++ and Java. In addition, we found examples of not so main stream teaching languages like Assembler [36, 43] and shell scripts [70].

Some of the systems are language independent. Especially if the assessment is based on output comparison, any language that can be executed on the same environment can be automatically assessed after the system is configured to compile and execute solutions in that language.

4.2 Learning Management Systems

Extending the existing learning management systems (LMS) like Moodle[4] to better fit into the special needs of CS education seems to draw increasing interest (e.g. [56, 62, 63], the forthcoming ITiCSE'10 working group, and many others). One argument supporting LMS–AA integration is to avoid reimplementing all the course management features. As a LMS hosting several (not only programming) courses has a huge number of users, it is a tempting target for attackers. Malicious code executed in such an environment is always a serious threat. Therefore, securing AA systems integrated into LMSs is extremely important. Despite the challenges, we believe that there are more pros than cons in this approach and that there is an increasing demand to bring automatically assessed exercises into LMSs.

We found the following AA extensions to LMSs: CTPracticals [27] to bring VHDL and Matlab exercises into Moodle, Automatic Grader [74] to assess students' Java assignments in Sakai[5], AutoGrader [29, 51] to support Java assignments in Cascade[6], WeBWorK-JAG [21, 22, 23] to bring automatically assessed Java exercises into WeBWorK[7], SISA-EMU [36] to provide Assembler programming assignments through Moodle, and finally VERKKOKE [2] to bring socket programming and routing into any LMS with SCORM support (e.g. Moodle). In addition, EduComponents [3, 4] brings programming exercises to Plone[8], a content management system, not a fully featured LMS.

[4] http://moodle.org/
[5] http://sakaiproject.org/
[6] http://www.cascadelms.org/
[7] http://webwork.maa.org/

4.3 Defining Tests

Assessing the functionality of students' code is still the most often used approach to grade programs. The ways to do this can be divided between the use of industrial testing tools and various specialized solutions. Examples of using industrial tools were:

- **XUnit** based approaches were used in several systems (e.g. [3, 72]). In some cases students even created their own tests with JUnit [15].
- **Acceptance testing frameworks**, (e.g. EasyAccept [66, 67]) where tests are defined in a natural-language-like scripting language. Tests are easy-to-read requirement specifications as well as used for the assessment at the same time.
- **Webtesting frameworks** like Watir[9] in AWAT [75] and Selenium[10] in Electronic Commerce Virtual Laboratory [11].

Specialized solutions included:

- **Output comparison** is the traditional approach used by many of the systems we found. Survey of Ala-Mutka [1] already reported several variations of output comparison including running the model solution and student's code side by side and the use of regular expressions to match the output.
- **Scripting** can mean almost everything, and at the same time it is the most commonly reported way to define tests. For example, a script can be a shell script compiling the program, running it, and comparing the output to a file containing the expected output.
- **Experimental approaches** like comparing program graphs of a student's solution to the pool of known correct answers [50, 80] or deriving test cases with a model checker [33] were also reported.

4.4 Resubmissions

Practice is important in learning programming and there should be room for mistakes and learning from them. AA can help as it can give feedback despite the limited human resources. However, to prevent mindless trial-and-error problem solving, the number of resubmissions should be controlled [44]. Here are some examples of how the problem of trial-and-error can be tackled.

- **Limiting the number of submissions**, in addition to having deadlines, is the trivial approach supported by most of the current systems.
- **Limiting the amount of feedback** is another classical way to force students think after a failed submission. However, this can also create confusion among students. Especially, students not familiar with AA (who do not trust AA yet) may feel that the feedback

[8] http://plone.org/
[9] http://watir.com/
[10] http://seleniumhq.org/

provided by the system is erroneous if they are not able to understand why their submission was judged wrong [26].

- **Compulsory time penalty** after each submission can be used to direct students behavior [1]. Moreover, length of this penalty can grow exponentially after each failed attempt [35].

- **Making each exercise slightly different** is an interesting concept used in QuizPACK by allowing parameterized, automatically assessed random assignments for C programming [8]. Trial-and-error makes no sense when you need to start from scratch to submit again.

- **Programming contests** provide a completely alternative approach where the assignment specification is visible only for a short period of time during which the assignment needs to be completed while competing against time (and others). This approach is adopted to education, for example, in Mooshak [26]. The competition aspect has been proven to be an excellent motivation for the students [41] but also generates a number of problems. How to teach students good scheduling of software development process if they are encouraged to perform as fast as possible at least partly regardless of the quality of the work?

- **Various hybrid approaches** and modifications are also possible. For example, Marmoset [72] supports both unlimited and limited number of submissions. First, there is a public test set to check the basic functionality. These tests can always be executed and repeating submissions are not penalized. Second, there are release tests that can only be asked n-times. Feedback from the release tests is also limited to force students to think before asking tests to be executed.

4.5 Possibility for Manual Assessment

It is often a good idea to combine both manual and automated assessment. Teaching assistants (TAs) can provide extra feedback by manually assessing a submission or they can override the grades, etc. From the tools we surveyed, we were able to identify two levels of manual intervention (no support for manual intervention being the third).

- **To enable the teacher to view the student submissions** is the lightest way to support manual intervention. In this approach, the tool itself does not provide any features for the marking but at least makes it possible to manually assess the same submission. Often the same effect can be achieved by logging into the assessment system and fetching the submissions from the database or filesystem where they are stored. However, supporting this through the AA system makes it possible to separate the roles of TAs from administrators of the AA system.

- **Combining manual and automatic feedback** means TAs feedback and automated assessment can both exist at the same time and support each others. This is supported in Web-Cat [16], for example.

None of the systems clearly described that they would allow TAs to completely override the automatic feedback but we still expect some systems to support this. However, this can easily create confusion among both teachers and learners if the origins of the grade are not transparent.

4.6 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 [18]. However, as important as this topic is, a large portion of the included articles ignored this. The following approaches to secure execution of students' code were mentioned in the articles:

- **Proper sandboxing.** Relying on existing solutions to securely run programs is a common approach. This can be done by using multiple tools like Systrace (used in EduComponents [3]), linux security module (used in [48]), Java security policy (used in [48]), `ptrace` (used in Moe [46]), and `chroot` (used in CTPracticals [27]).

- **Static analysis.** Security can also be addressed by using custom solutions. For example, Algorithm Benchmark uses regular expressions to try to filter out malicious code [10].

- **Grading on the client.** Some systems deal with sandboxing by doing the grading on the client side in students' own computers. *Mailing It In* [65] uses client's email software to launch tests on client side, whereas E-Commerce virtual laboratory [11] uses Selenium-RC to push tests back to the client that did the submission.

Additional security feature implemented in some systems is to have a different server for running the student programs instead of doing it all on the same machine as the rest of the system. This is done, for example, by EduComponents [3].

In addition to securing AA systems, sandboxing can help when assessing the performance of students' programs. Sandboxes can be configured to limit the available memory or CPU time to ensure assessed solutions are efficient enough (e.g. [27, 79]).

4.7 Distribution and Availability

It is surprising, and quite disappointing, to see how few systems are open-source, or even otherwise (freely) available. In many papers, it is stated that a prototype was developed but we were not able to find the tool. In some cases, a system might be mentioned to be open source but you need to contact the authors to get it.

4.8 Specialty

Quite often the driving force for the development of a completely new tool is a revolutionary idea of something that has not yet been done. Or at least this is the case with tools that get researched and published. Specialities of AA systems identified during the survey included:

- **Automatic assessment of GUIs** has been identified already in the survey of Douce [14] and is still of interest. New systems are still developed [24] and the existing ones are extended to meet the special requirements of GUI exercises [77].

- **SQL** tutoring systems have existed since the late 90's. New systems for this specialty were recognized also in this survey (e.g. [13, 38])

- **Concurrent programming** assignments are often extremely error prone and problems may be hard to detect. Testing concurrency is demanding and specialized tools are developed to help (e.g. [52]).

- **Web-programming** and testing both functionality and security of the websites students implement is getting more attention together with the web-programming getting a stronger position in the curricula. These systems are typically testing a web site (HTML + JavaScript) ignoring how the server side of the site is implemented (e.g. [11, 19, 28, 32, 75]).

- **Letting students do the quality assurance**, either by writing tests for themselves (e.g. [77]) or reviewing code of others (e.g. [61]) is often well grounded to the pedagogical needs.

5. DISCUSSION AND CONCLUSIONS

In this paper, we have surveyed the recent developments of automatic assessment tools for programming assignments. We have done this by systematically collecting relevant articles published in years 2006-2010 to get a sense what has happened in the field since the previous literature reviews on the topic were conducted. The systems included can be roughly divided into two categories: 1) automatic assessment systems for programming competitions and 2) automatic assessment systems for (introductory) programming education.

To answer our first research questions, we have discussed the key features of AA systems in Section 4. From these, we think that the differences in how tests are defined, how resubmissions are handled, and how the security is guaranteed were the most significant.

Based on the data we collected, it is possible to make some recommendations how new AA systems could get more widely adopted. First, we recommend that authors describing new systems should explain more explicitly how the system actually works and provide more examples. For example, instead of stating that the assessment is based on scripts, examples should be provided.

In addition, security of the assessment systems should get more attention. Use of proper sandboxing based on existing security solutions should be encouraged and use of home baked static analysis should be avoided. The latter can leave the system vulnerable, since, for example, the filtering of code using regular expressions is error-prone. Ultimately the security needs to be provided in a way that makes installing the system as easy as possible without compromising security. However, configuring a sandbox can be complicated. Preferably, initial security configuration should not rely on teachers' skills. For example, writing a proper Java security policy is doable (although letting AA system to provide such policies is better) but setting up a secure linux playground with `chroot`, for example, is demanding and teachers might be tempted to make shortcuts. In fact, we believe that lack of sandboxing, or the difficulties in configuring the sandbox, is often one of the obstacles in adopting a system.

The lack of open-source systems might be one of the reasons for the constant development of new tools – that are also likely to remain in-house. We understand people do not want to publish something unfinished but at the same time this slows down new ideas from spreading wider. Thus, we argue that by open-sourcing the existing tools to some popular online version control repository like GitHub[11] or Google Code[12], the tools could be much more willingly adopted by others.

To answer our second research question, we expect new research to emerge from the following fields, from where the first steps were identified in this survey:

- Integrating automatic assessment into LMSs. As another possible path, some of the assessment systems can grow into LMSs if they are modular enough and if they get the momentum behind them. For example, we see that Web-CAT with the various assessment modules already implemented into it is a good candidate to become a CS specific LMS.

- Putting more effort into security of automatic assessment systems. This is also related to the LMS integration because having multiple courses hosted on one platform makes this a more tempting target to hack.

- Automatic assessment of web applications students implement. This can be seen to continue the GUI and SQL testing efforts that have longer traditions. The new aspect we expect to get more importance is security/penetration testing of students' web applications.

There seems to be a steady interest in developing new automatic assessment tools. Sometimes the need to implement yet another system can be challenged and one should ask whether the new feature could be added directly into an existing open source system as in Web-CAT [77], for example. In addition, to increase the adoption of existing systems and thus avoiding the reinvention of the wheel, we strongly suggest automatic assessment system developers to make their systems open source making it easier for others to contribute.

6. FUTURE WORK

Classification presented in Section 4 can be further improved. For example this could be a starting point for a more formal Delphi study [64] with more experts deciding on the categories. Outcome could result in a taxonomy on automatic assessment of programming assignments.

In this survey, we had quite narrow scope. There are many systems closely related to AA we did not cover: systems designed for formative/visual feedback (e.g. [31]), peer review systems (e.g. [71]), and systems to provide feedback on misconceptions and problem solving strategies (e.g. [59]) – to name a few. Identifying the types of systems that can cooperate in an AA setup is essential for understanding how AA systems should be improved from the technical perspective.

Many of the papers we surveyed reported educational experimentations and results of comparing different approaches in automatic assessment. Combining those results with the features of AA systems (presented in this paper) is something we are looking next. This is important for improving AA systems from the pedagogical perspective.

[11] `http://github.com/`
[12] `http://code.google.com`

7. REFERENCES

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

[2] A. Alstes and J. Lindqvist. Verkkoke: learning routing and network programming online. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 91–95, New York, NY, USA, 2007. ACM.

[3] M. Amelung, P. Forbrig, and D. Rösner. Towards generic and flexible web services for e-assessment. In *ITiCSE '08: Proceedings of the 13th annual Conf. on Innovation and technology in computer science education*, pages 219–224, New York, NY, USA, 2008. ACM.

[4] M. Amelung, M. Piotrowski, and D. Rösner. Educomponents: experiences in e-assessment in computer science education. In *ITICSE '06: Proceedings of the 11th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 88–92, New York, NY, USA, 2006. ACM.

[5] D. J. Barnes and M. Kölling. *Objects First with Java - A Practical Introduction using BlueJ, Thid edition*. Prentice Hall / Pearson Education, 2006.

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

[7] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.*, 80(4):571–583, 2007.

[8] P. Brusilovsky and S. Sosnovsky. Individualized exercises for self-assessment of programming knowledge: An evaluation of quizpack. *J. Educ. Resour. Comput.*, 5(3):6, 2005.

[9] J. Carter, J. English, K. Ala-Mutka, M. Dick, W. Fone, U. Fuller, and J. Sheard. ITICSE working group report: How shall we assess this? *SIGCSE Bulletin*, 35(4):107–123, 2003.

[10] M.-Y. Chen, J.-D. Wei, J.-H. 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 Conf. on Innovation and technology in computer science education*, pages 123–127, New York, NY, USA, 2006. ACM.

[11] J. Coffman and A. C. Weaver. Electronic commerce virtual laboratory. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, pages 92–96, New York, NY, USA, 2010. ACM.

[12] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *CCSC '00: Proceedings of the fifth annual CCSC northeastern Conf. on The journal of computing in small colleges*, pages 107–116, , USA, 2000. Consortium for Computing Sciences in Colleges.

[13] M. de Raadt, S. Dekeyser, and T. Y. Lee. Do students sqlify? improving learning outcomes with peer review and enhanced computer assisted assessment of querying skills. In *Proceedings of the 6th Baltic Sea Conf. on Computing education research*, pages 101–108, New York, NY, USA, 2006. ACM.

[14] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4, 2005.

[15] S. H. Edwards and M. A. Pérez-Quiñones. Experiences using test-driven development with an automated grader. *J. Comput. Small Coll.*, 22(3):44–50, 2007.

[16] S. H. Edwards and M. A. Pérez-Quiñones. Web-cat: automatically grading programming assignments. In *ITiCSE '08: Proceedings of the 13th annual Conf. on Innovation and technology in computer science education*, pages 328–328, New York, NY, USA, 2008. ACM.

[17] M. Forišek. On the suitability of programming tasks for automated evaluation. *Informatics in education*, 5(1):63–76, 2006.

[18] M. Forišek. Security of programming contest systems. In *Informatics in Secondary Schools, Evolution and Perspectives*, Vilnius, Lithuania, 2006.

[19] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu. Apogee: automated project grading and instant feedback system for web based computing. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 77–81, New York, NY, USA, 2008. ACM.

[20] G. Gárcia-Mateos and J. L. Fernández-Alemán. A course on algorithms and data structures using on-line judging. *SIGCSE Bull.*, 41(3):45–49, 2009.

[21] O. Gotel and C. Scharff. Adapting an open-source web-based assessment system for the automated assessment of programming problems. In *WBED'07: Proceedings of the sixth Conf. on IASTED International Conf. Web-Based Education*, pages 437–442, Anaheim, CA, USA, 2007. ACTA Press.

[22] O. Gotel, C. Scharff, and A. Wildenberg. Extending and contributing to an open source web-based system for the assessment of programming problems. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 3–12, New York, NY, USA, 2007. ACM.

[23] O. Gotel, C. Scharff, and A. Wildenberg. Teaching software quality assurance by encouraging student contributions to an open source web-based system for the assessment of programming assignments. *SIGCSE Bull.*, 40(3):214–218, 2008.

[24] G. R. Gray and C. A. Higgins. An introspective approach to marking graphical user interfaces. In *ITICSE '06: Proceedings of the 11th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 43–47, New York, NY, USA, 2006. ACM.

[25] T. Greening. Computer Science Educational Futures: The Nature of 2020" Foresight. In *Computer Science Education in the 21st Century*, pages 1–6. Springer Verlag, 1999.

[26] P. Guerreiro and K. Georgouli. Combating anonymousness in populous cs1 and cs2 courses. In *ITICSE '06: Proceedings of the 11th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 8–12, New York, NY, USA, 2006. ACM.

[27] E. Gutiérrez, M. A. Trenas, J. Ramos, F. Corbera, and S. Romero. A new moodle module supporting automatic verification of vhdl-based assignments. *Comput. Educ.*, 54(2):562–577, 2010.

[28] F. Gutierrez. Stingray: a hands-on approach to learning information security. In *SIGITE '06: Proceedings of the 7th Conf. on Information technology education*, pages 53–58, New York, NY, USA, 2006. ACM.

[29] M. T. Helmick. Interface-based programming assignments and automatic grading of java programs. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 63–67, New York, NY, USA, 2007. ACM.

[30] C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas. The coursemarker cba system: Improvements over ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

[31] C. D. Hundhausen and J. L. Brown. An experimental study of the impact of visual semantic feedback on novice programming. *J. Vis. Lang. Comput.*, 18(6):537–559, 2007.

[32] W.-Y. Hwang, C.-Y. Wang, G.-J. Hwang, Y.-M. Huang, and S. Huang. A web-based programming learning environment to support cognitive development. *Interacting with Computers*, 20(6):524 – 534, 2008.

[33] P. Ihantola. Creating and visualizing test data from programming exercises. *Informatics in education*, 6(1):81–102, 2007.

[34] D. Jackson and M. Usher. Grading student programs using assyst. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, New York, NY, USA, 1997. ACM.

[35] T. Janhunen, T. Jussila, M. Järvisalo, and E. Oikarinen. Teaching Smullyan's analytic tableaux in a scalable learning environment. In *Proceedings of the Fourth Finnish / Baltic Sea Conf. on Computer Science Education*, volume TKO-42/04 of *Research Report Series of Laboratory of Information Processing Science, Helsinki University of Technology*, pages 85–94. Otamedia, December 2004.

[36] D. Jimenez-Gonzalez, C. Alvarez, D. Lopez, J.-M. Parcerisa, J. Alonso, C. Perez, R. Tous, P. Barlet, M. Fernandez, and J. Tubella. Work in progress-improving feedback using an automatic assessment tool. In *Proceedings of 38th annual Frontiers in Education Conf.*, pages S3B–9 –T1A–10, oct. 2008.

[37] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. In *ACM Journal on Educational Resources in Computing, volume 5, number 3, September 2005. Article 2.* ACM, 2005.

[38] H. Ke, G. Zhang, and H. Yan. Automatic grading system on sql programming. In *Scalable Computing and Communications; Eighth International Conf. on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conf. on*, pages 537 –540, sept. 2009.

[39] R. Kolstad. Infrastructure for contest task development. *Olympiads in Informatics*, 3:38–59, 2009.

[40] A. Korhonen, L. Malmi, and P. Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of the Third Annual Baltic Conf. on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[41] T. Lehtonen. Javala – addictive e-learning of the java programming language. In *Koli Calling 2005 – Fifth Koli Calling Conf. on Computer Science Education*, pages 41–48, 2005.

[42] Y. Liang, Q. Liu, J. Xu, and D. Wang. The recent development of automated programming assessment. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conf. on*, pages 1 –5, dec. 2009.

[43] M. Lingling, Q. Xiaojie, Z. Zhihong, Z. Gang, and X. Ying. An assessment tool for assembly language programming. In *Computer Science and Software Engineering, 2008*, volume 5, pages 882 –884, dec. 2008.

[44] L. Malmi, V. Karavirta, A. Korhonen, and J. Nikander. Experiences on automatically assessed algorithm simulation exercises with different resubmission policies. *Journal of Educational Resources in Computing*, 5(3), September 2005.

[45] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 297–306, New York, NY, USA, 2006. ACM.

[46] M. Mareš. Perspectives on grading systems. *Olympiads in Informatics*, 1:124–130, 2007.

[47] M. Mareš. Moe – design of a modular grading system. *Olympiads in Informatics*, 3:60–66, 2009.

[48] B. Merry. Using a linux security module for contest security. *Olympiads in Informatics*, 3:67–73, 2009.

[49] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conf. on Advanced Visual Interfaces*, pages 373 – 376, Gallipoli (Lecce), Italy, May 2004. ACM.

[50] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking student programs using graph similarity. *Comput. Educ.*, 54(2):545–561, 2010.

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

[52] R. Oechsle and K. Barzen. Checking automatically the output of concurrent threads. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 43–47, New York, NY, USA, 2007. ACM.

[53] J. O'Kelly and J. P. Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *ITICSE '06: Proceedings of the 11th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 217–221, New York, NY, USA, 2006. ACM.

[54] R. E. Pattis, J. Roberts, and M. Stehlik. *Karel the robot (2nd ed.): a gentle introduction to the art of programming*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[55] A. Pears, S. Seidman, C. Eney, P. Kinnunen, and L. Malmi. Constructing a core literature for computing education research. *SIGCSE Bulletin*, 37(4):152–161, 2005.

[56] A. Radenski. Digital cs1 study pack based on moodle and python. In *ITiCSE '08: Proceedings of the 13th annual Conf. on Innovation and technology in computer science education*, pages 325–325, New York, NY, USA, 2008. ACM.

[57] M. A. Revilla, S. Manzoor, and R. Liu. Competitive learning in informatics: The uva online judge experience. *Olympiads in Informatics*, 2:131–148, 2008.

[58] P. Ribeiro and P. Guerreiro. Increasing the appeal of programming contests with tasks involving graphical user interfaces and computer graphics. *Olympiads in Informatics*, 1:139–164, 2007.

[59] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and behavioral predictors of novice programmer achievement. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE Conf. on Innovation and technology in computer science education*, pages 156–160, New York, NY, USA, 2009. ACM.

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

[61] G. Rößling and S. Hartte. Webtasks: online programming exercises made easy. *SIGCSE Bull.*, 40(3):363–363, 2008.

[62] G. Rößling, M. Joy, A. Moreno, A. Radenski, L. Malmi, A. Kerren, T. Naps, R. J. Ross, M. Clancy, A. Korhonen, R. Oechsle, and J. A. V. Iturbide. Enhancing learning management systems to better support computer science education. *SIGCSE Bull.*, 40(4):142–166, 2008.

[63] G. Rößling and T. Vellaramkalayil. A visualization-based computer science hypertextbook prototype. *Trans. Comput. Educ.*, 9(2):1–13, 2009.

[64] G. Rowe and G. Wright. The delphi technique as a forecasting tool: issues and analysis. *International Journal of Forecasting*, 15(4):353–375, October 1999.

[65] J. A. Sant. "mailing it in": email-centric automated assessment. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE Conf. on Innovation and technology in computer science education*, pages 308–312, New York, NY, USA, 2009. ACM.

[66] J. P. Sauvé and O. L. Abath Neto. Teaching software development with atdd and easyaccept. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 542–546, New York, NY, USA, 2008. ACM.

[67] J. P. Sauvé, O. L. Abath Neto, and W. Cirne. Easyaccept: a tool to easily create, run and drive development with automated acceptance tests. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 111–117, New York, NY, USA, 2006. ACM.

[68] J. Sheard, S. Simon, M. Hamilton, and J. 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, New York, NY, USA, 2009. ACM.

[69] M. Sitaraman, J. O. Hallstrom, J. White, S. Drachova-Strang, H. K. Harton, D. Leonard, J. Krone, and R. Pak. Engaging students in specification and reasoning: "hands-on" experimentation and evaluation. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE Conf. on Innovation and technology in computer science education*, pages 50–54, New York, NY, USA, 2009. ACM.

[70] A. Solomon, D. Santamaria, and R. Lister. Automated testing of unix command-line and scripting skills. In *Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th International Conf. on*, pages 120 –125, july 2006.

[71] H. Sondergaard. Learning from and with peers: the different roles of student peer reviewing. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE Conf. on Innovation and technology in computer science education*, pages 31–35, New York, NY, USA, 2009. ACM.

[72] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. 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 Conf. on Innovation and technology in computer science education*, pages 13–17, New York, NY, USA, 2006. ACM.

[73] J. T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[74] H. Suleman. Automatic marking with sakai. In *SAICSIT '08: Proceedings of the 2008 annual research Conf. of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, pages 229–236, New York, NY, USA, 2008. ACM.

[75] M. Sztipanovits, K. Qian, and X. Fu. The automated web application testing (awat) system. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conf.*, pages 88–93, New York, NY, USA, 2008. ACM.

[76] P. G. Thomas, N. Smith, and K. G. Waugh. Computer assisted assessment of diagrams. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE Conf. on Innovation and technology in computer science education*, pages 68–72, New York, NY, USA, 2007. ACM.

[77] M. Thornton, S. H. Edwards, R. P. Tan, and M. A. Pérez-Quiñones. Supporting student-written tests of gui programs. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 537–541, New York, NY, USA, 2008. ACM.

[78] D. 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, New York, NY, USA, 2004. ACM Press.

[79] T. Verhoeff. Programming task packages: Peach exchange format. *Olympiads in Informatics*, 2:192–207, 2008.

[80] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. *Inf. Softw. Technol.*, 49(2):99–107, 2007.

APPENDIX

A. LIST OF TOOLS

This appendix briefly describes publicly available (to download or with a demo site where to experiment) tools found in this survey. Many papers did not have explicit statement about the availability of the system, and in many cases we failed to find a site. There were also cases where university's public version control where the system was distributed no longer existed. It should be noted that some of the tools we failed to find could be available by asking the authors. This was not done. References found in this survey and the URL from where more information can be retrieved are mentioned for each system.

AutoGrader [29, 51] is a subproject of Cascade LMS. It has been used to assess Java, but according to authors it can be extended to other languages. Tests are executed through Java reflection similarly to what JUnit does.
`http://www.cascadelms.org/autograder/` (GPL like)

AWAT [75] is an environment for web programming assignments where students only submit an URL of a site they developed. Teacher defines which components should exist on the web-page and tests by using the Watir Ruby library both combined into an Excel sheet. Testing is then performed by using Internet Explorer from the submission server.
Open source, contact authors

CTPracticals [27] is a Moodle module to bring automatically assessed VHDL exercises into Moodle. External test script and sandboxing are both configurable through the Moodle UI. The framework can also be extended to other programming languages. For example, there are Matlab exercises on the demo site.
`http://guac.ac.uma.es/demo` (login credentials in [27])

EasyAccept [66, 67] framework provides a natural-language-like scripting language to write tests for Java programs. Requirements are presented in a form of acceptance tests.
`http://easyaccept.sourceforge.net/` (GPL)

EduComponents [3, 4] is a set of components to the Plone CMS for creation, management and assessment of programming assignments. It has different backends for different programming languages which allow (depending on the language) unit testing, comparison to a model answer or more formally defined testing.
`http://plone.org/products/ecautoassessmentbox/` (GPL)

Linuxgym [70] supports exercises and examinations of unix scripting skills. An extensive exercise definition language is also included.
`http://linuxgym.com/` (GPL)

Moe [46, 47] (originally MO-eval) is a modular environment for programming contests with sandbox, queue manager, and submitter for managing submissions, and different graders (that can be combined). The aim is make various modules interchangeable.
`http://mj.ucw.cz/moe/` (GPL2)

Mooshak [20, 26, 58] has its origins in programming contests, although it has also been used in teaching. One of the specialties of Mooshak is that results of the assessment can be publicly shown to other students.
`http://code.google.com/p/mooshak/` (Artistic License/GPL)

Peach[3] [79] is a highly configurable system for programming education and contest hosting.
`http://peach3.nl/` (Artistic License v2)

ProtoAPOGEE [19] is a prototype of a proposed system to grade web sites like AWAT. APOGEE relies on Watir test library and it can also take series of screenshots from the web site being assessed. Screenshots can then be used as feedback to explain why a test failed.
`http://vlab.gsw.edu/Projects/APOGEE/` (sources and video for academic research or evaluation)

Resolver [69] combines formal verification and traditional programming assignments. There are exercises where students need to demonstrate their understanding of formal specifications by writing tests and exercises where students write programs verified against formally expressed contracts.
`http://www.cs.clemson.edu/~resolve/` (GPL3)

RoboCode [53] supports Java and .NET assignments where students' programs compete with each other. Grades can be based on the results of the competition.
`http://robocode.sourceforge.net/` (Eclipse Public License)

USACO's [39] competition hosting environment has been developed by the USA Computing Olympiad. The system also offers web based problem development tools to aid in creating competition problems.
`http://train.usaco.org/usacogate` (demo)

UVA Online Judge [57] is mainly intended as a programming contest training site. Users can practice on the large number of existing problems and submit their answers in multiple languages. It is also used for hosting online programming competitions.
`http://uva.onlinejudge.org` (demo)

VERKKOKE [2] is an online teaching environment for socket programming/routing. It generates individual programming assignments which the student completes and submits. One of the specialties is the SCORM integration with LMS systems (e.g. Moodle and Optima).
`http://www.tml.tkk.fi/Research/VERKKOKE/` (MIT)

Web-CAT [15, 77] is a system where students are required not only to submit source code, but also unit test their own code. Part of the grade is based on the test coverage achieved by students' own tests. Web-CAT has a plugin architecture for different graders, static analysis, support for other languages, etc.
`http://web-cat.cs.vt.edu/WCWiki` (Affero GPL)

WeBWorK-JAG (Java auto-grader) [21, 22, 23] is an extension module to the WeBWorK exercise delivery platform. The module allows checking Java programs with JUnit.
`http://csis.pace.edu/~scharff/webwork`

Publication (vi)

Ahoniemi, T., Lahtinen, E., Reinikainen, T. Improving Pedagogical Feedback and Objective Grading *In: The Proceedings of SIGCSE'08*, March 2008, Portland, Oregon, USA.

Improving Pedagogical Feedback and Objective Grading

Tuukka Ahoniemi
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
tuukka.ahoniemi@tut.fi

Essi Lahtinen
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
essi.lahtinen@tut.fi

Tommi Reinikainen
Institute of Software Systems
Tampere University of Technology
Tampere, Finland
tommi.reinikainen@tut.fi

ABSTRACT

It is important for learning that students receive enough of educational feedback of their work. To get the students to be seriously disposed to the feedback it has to be personal, objective and consistent. In large classes ensuring such feedback can be difficult. Grading rubrics are a solution to the objectivity and consistency.

ALOHA is an online grading tool based on rubrics which all the graders have to use. Particularly, ALOHA provides features that make the grading process more convenient for the graders and the teacher. By facilitating the graders work ALOHA allows them to focus more on feedback writing.

To test the effectiveness of ALOHA in objectivity and consistency we did a comparative statistical analysis on the distribution of grades. The results supported the assumptions showing improvement resulting in similar distribution of grades amongst different graders who used the tool.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Algorithms, Design, Human Factors

Keywords

Grading; Rubrics; Programming; Assessment; Teaching

1. INTRODUCTION

Novice level programming courses in Tampere University of Technology tend to be large courses with hundreds of students. On one hand we need to provide profound and personalized feedback for novice students to enhance learning but on the other hand we need to keep our workload maintainable.

Computer-aided assessment (CAA) is a way to facilitate the workload [5]. Its use can be divided into fully automatic assessment and semi-automatic assessment. Using fully automatic assessment is efficient but can not be used for everything. The novice students need feedback from a human being in things like design and coding style. [3]

Semi-automatic assessment is a compromise between automatic and manual assessment: automatic tools are used for some parts of the grading and manual assessment for others [2]. As many good tools like BOSS, ASSYST, Ceilidh, Scheme-Robo, and CourseMarker [5] exist for automatically assessing the program correctness, our work focuses on facilitating the manual assessment, which usually requires the use of multiple graders in mass courses.

Our object was to facilitate the manual assesment process by automating its repetitive routines and to achieve consistency and objectivity between graders. We present general features and ideas implemented in a grading tool called ALOHA, briefly introduced in our earlier work [1]. We will also present a statistical analysis regarding the benefits of the tool from the perspective of consistency and objectivity of the grades.

2. PROBLEMS OF GRADING

One problem with large courses is that many student submission resemble each other. As the grader is supposed to inspect dozens of these submissions it is likely that without care the objectivity would suffer. For instance his mood could affect the grading as well as the order in which he is grading the submissions—a mediocre submission can appear brilliant if it preceeds a couple of lousy ones. Also the grader can in a rush forget to check some parts of the work for some submission resulting in biases in grading.

According to Habeshaw et al. [7], the only definite way to grade objectively is by using only objective tests (multiple-choice questions etc.), but doing so would not be desirable in courses where a major part of the learning is based on the student programming by himself. To ensure consistency among graders, they need to be properly trained for the job and they are required to use marking schemes to direct their attention to the appropriate things in a student's work [7]. Becker suggests a similar approach and also defines the schemes as *rubrics* [4].

The use of rubrics increases objectivity because the assessment is split into small enough parts. The grader can concentrate on a single aspect of the work instead of giving a general grade for the final work. The total grade can later be derived from the grades given to the parts.

As many of the submissions are alike they also have the same mistakes and the grader keeps repeating the same im-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'08, March 12–15, 2008, Portland, Oregon, USA.
Copyright 2008 ACM 978-1-59593-947-0/08/0003 ...$5.00.

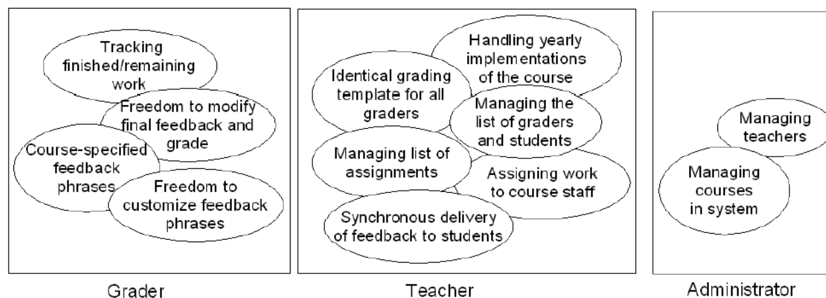


Figure 1: The user roles in ALOHA and their main functionalities.

provement ideas over and over again. It is convenient for the grader that he can easily reuse the feedback phrases he has written earlier. However, the feedback should not be generic multipurpose phrases but detailed, personal comments pointing out particular sections of that work. The feedback should help the students to improve their work or to suggest what to do differently.

3. DESCRIPTION OF ALOHA TOOL

Earlier we used traditional rubrics in grading but unlike Becker we did not let the students see the filled rubrics. All graders did not use the rubrics properly but might have formed the grade before even filling the rubric. The purpose of ALOHA[1] is to provide the rubrics online in a way that each grader has to fill them correctly and in a coherent way. ALOHA also helps the grader in creating feedback by providing the opportunity to use, edit and personalize ready-written phrases.

ALOHA is run on a separate server and is used through a web browser (resulting in platform indepency). Each user has access to certain roles in the system: administrator, teacher and grader. The borders between the different user roles are absolute, meaning that if a teacher wishes to grade a work, he must first have the appropriate role. This is done to distinguish the different user types from one another and to ensure that only the grader has the power to give the grade. The main functionality for each role is represented in Figure 1.

The teacher is responsible for adding graders and students to a course and assigning student submissions for the graders. The rubric is defined by creating an XML file that defines the structure the grading process should follow. It also defines some template feedback phrases for the graders. Finally, after a submission is graded, the teacher can send the related feedback to the students or student groups either individually or all-at-once. The only role that has the ability to actually send the feedback is the teacher.

3.1 Grading with ALOHA

After the grader has logged in to the tool, he is presented with a list of student submissions the teacher has assigned to him. This is called the *work list.* Each submission has a state which describes whether the grading of the submission is yet to be started, in an unfinished state or finished and accepted. The grader chooses one of the submissions on his work list and is taken into the grading view, shown in Figure 2.

[1]Arvostelutyökalu Laitoksen Ohjelmointikurssien HArjoitustöille

3.1.1 Grading Hierarchy

The grading view represents the grading rubric. The grading is based on a hierarchy shown in the left of Figure 2 consisting of *categories* (in the Figure: Documentation, Dynamic tests, etc.) Each category is divided into *subcategories* (First document, Final document, etc.) These are given a grade based on the *grading items* related to it (”First document returned?”, etc.) Each grading item has a collection of phrase templates that the grader can add to the feedback text of that subcategory and personalize.

3.1.2 Semi-Automatic Phrasing

The phrase templates form one of the main features in ALOHA called *semi-automatic phrasing.* This means that the grader can choose suitable phrases to be added directly to the feedback text. This is an idea which was also found useful by the creators of Agar [10]. When a phrase is selected, it is copied as text into one of the Positive/Negative/ Neutral -feedback forms, depending on the type of the phrase (each phrase is classified as one). The text in the form is editable so the grader may modify it if he wishes. Often the idea is that the phrase the grader chooses from the list of phrases is purposefully incomplete and the grader manually completes the phrase, thus personalizing the feedback.

3.1.3 Forming the Grade

After all the grading items in one page have been given a value and the grader is satisfied with the feedback text, he must finish the grading of the subcategory by selecting a grade for that subcategory. The grader does this by looking at the values selected for each grading item. The item values do not force any certain grade. Their purpose is to show the grader how well the submission faired in this subcategory. To help in selecting the appropriate grade, the teacher may have provided a hint for that subcategory.

ALOHA uses the grade of the subcategory to award the submission a number of points for that particular subcategory. The relationship between the grade and the amount of points is defined by the teacher while configuring the assignment. After the grade has been given, the grading of this subcategory is finished and that subcategory is marked as ready.

The grader continues to grade the submission subcategory by subcategory until he has finished them all. At this point, the grader is offered an option to finish the grading of the entire submission. By selecting to finish the grading, he is taken to preview the final feedback and shown the final grade suggestion. The final feedback is composed from all the individual phrases the grader chose and personalized or wrote himself during grading the subcategories. The final

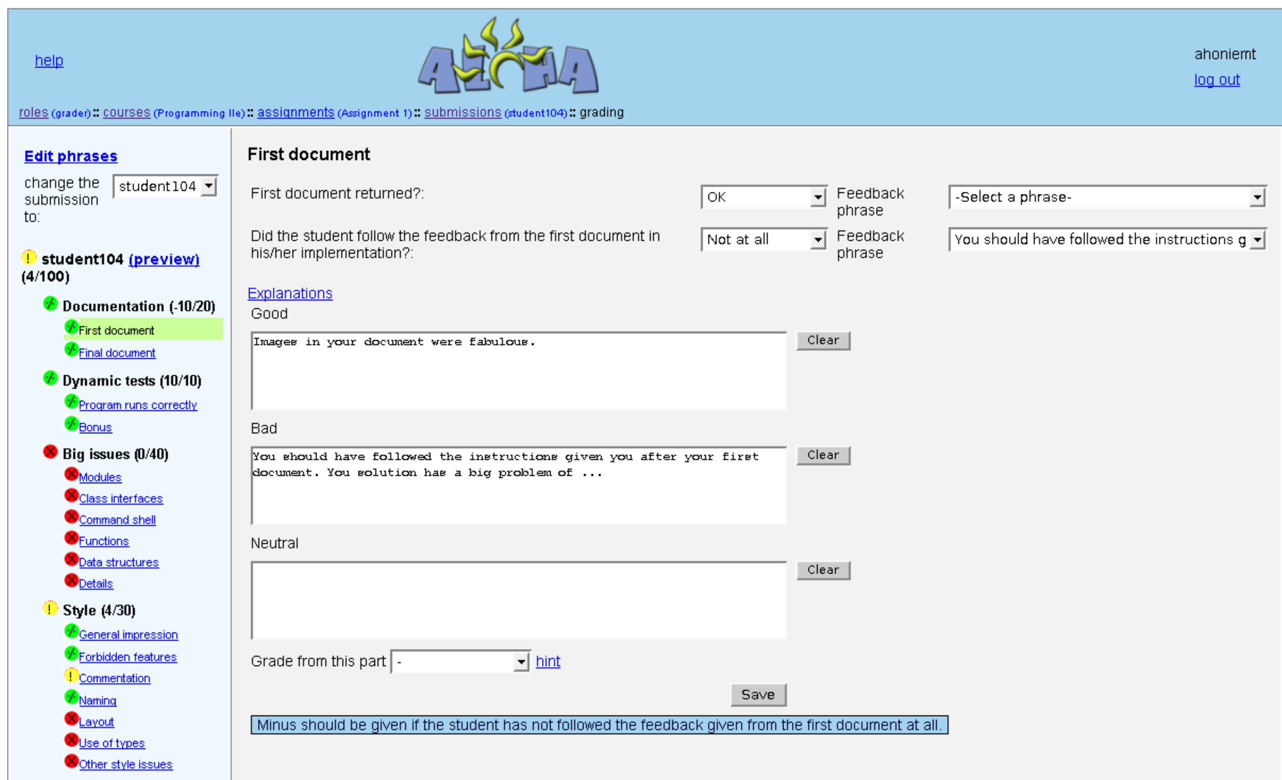


Figure 2: Screenshot of the grading view in ALOHA

grade suggestion is similarly calculated by using the individual grades given to each subcategory. The calculation formula defined by the teacher takes the weighting of subcategories into account.

3.2 Customization

ALOHA offers some important customization features. For the teachers, it offers a chance to define the grading scale individually for each assignment. This is done by implementing an assignment-specific plugin that defines the grades, possible sanctions for submitting late etc. by each grading automatically. The plugin is implemented in PHP, just like the rest of the system. Instead of just plain grade limits based on points the more complex plugin-based solution allows more freedom for the teacher. The categories of the rubric can also be given different weightings in the XML file. Certain graded aspects of the submissions can be defined more valuable than others (e.g. programming style could be weighted more important than the functionality of the program).

For graders, ALOHA gives a lot of freedom in grading student work. ALOHA suggests a final grade based on the points given by the grader and the teacher-defined configuration of the assignment. The grader may then choose to accept the suggested grade as the final grade or to change it to any other possible grade (with explanation).

Another important feature for graders is the possibility to edit the phrases offered by the teacher through the assignment configuration, as well as defining their own phrases. This functionality exists because the teacher cannot preempt all possible phrases needed to grade a student work.

Finally, the grader is can edit the final feedback, which is eventually sent to students. This is not restricted so the grader has free hands to do what ever is most educative.

4. FIRST EXPERIENCES

This grading process was introduced during spring semester 2006 on four courses in TUT. Two of the courses were introductory course on programming and others a course on mobile programming and a course on advanced object-oriented programming.

4.1 User Opinions

The general opinion towards the process and the tool amongst the graders was positive. They liked to think about ALOHA as a check list for the important issues and the grade suggestion was found useful. The best-liked features were the semi-automatic phrasing and creating personalized feedback mails while filling out the rubrics.

If the grading scale was narrow (e.g. 0-3) the grade suggestion of ALOHA was not found as useful as when the scale was broader (e.g. 0-6). This is obvious because the grader can "see" if a submission is to be given 2 or 3 much easier in a narrow scale than in a broad scale.

Some graders had used the predecessor tool of ALOHA–Arvostin–and found the grader's customization possibilities described in Section 3.2 as a good add to an already good basic idea. Only one long-term grader found the process useless and said: "It took me much more time to grade, because I *had* to go through all the grading items". As a teacher who wants a consistent grading, this comment actually sounds very positive towards the tool.

The teachers liked the tool because of its benefits for the grader but they also saw the unpleasant side of the customization possibilities: it required much more effort to create the grading for a new assignment because of the large XML-template, submission listings and implementing the grading plugin. Still, this was the first time the tool was used and learning took time too. On the other hand teachers

found themselves saving time because of the administration features of ALOHA.

No profound study of student satisfaction towards feedback by ALOHA has been conducted. This is mostly because the tool has been used so far mainly in elementary programming courses so the students have not received any different sort of feedback for comparison. Nevertheless the students have said the feedback was useful for them for further courses.

4.2 Statistical analysis of the Objectivity

We compared the grading distributions of the same course, Programming 2, in two years. In both years there were nine graders. The assessment criteria were the same in both years. The grading scale was from zero to six points. In addition to program functionality the graders evaluated programming style, design and documentation. The submissions were divided amongst the graders randomly.

In the first year (2003) the grading was done without any tool. The graders were given similar rubrics to the ones ALOHA uses but printed in paper. No one could follow if the graders used them. In the second year (2006) the graders used ALOHA in the grading. We analyzed statistically only the submissions carried out without any special arrangements (165 students in 2003 and 109 in 2006). For the 2006 data we analyzed the grades ALOHA suggested. Possible changes in the final grades are not taken into account. The grades in 2003 were decided only by the grader himself.

Figure 3 shows the distribution, mean, and standard deviation of the grades of all nine graders in 2003. Figure 4 shows the same information from year 2006.

The mean values of the grades given by different graders were analysed by variance analysis (One-way ANOVA). The distributions of the grades given by all graders can be estimated to be close enough to normal distribution so that variance analysis can be used.

In the group of graders using ALOHA there were no statistically significant differences between the graders ($p > 0.05$). When the tool was not used there is a statistically significant difference ($p < 0.05$) between the grades given by the most lenient grader ("R" in Figure 3) and the ones given by the two strictest graders ("O" and "V").

5. DISCUSSION

The statistical analysis shows improvement in grading objectivity resulting in statistically insignificant differences amongst the graders. On the basis of only one comparison between two years this result cannot be generalized so that the use of online rubrics tool would be a solution for objective grading. A more sound proof would require statistics on inter-grader reliability using a common set of assignments. This would however require the extra resources to have multiple graders to grade the same submissions.

When not using ALOHA, most of the graders had graded coherently. The problems were limited to only a couple of graders whose grading was stricter or more lenient. On the basis of these results the tool seems to get these stricter or lenient graders to use similar distribution of grades than the others.

Other rubric-based assesment tools exist, but most of the research carried out on them is about peer-reviewing [8, 9] and some even on self-evaluation [6]. Thus many of the existing tools are designed so that they could be used by the students also. The tools, like RRAS [9] or Aropä [8], do not have features to facilitate the creation of the written feedback, like the successful phrasing feature of ALOHA. By leaving the possibility of students' own use out, we have been able to provide more features to aid the course staff. Thus, we should be able to facilitate our graders' work and give students more pedagocically valuable feedback on their work. A CAA-tool called Agar [10] has a similar idea but unlike Agar, ALOHA concentrates only on the manual grading instead of providing also automated tests and thus differs from ALOHA in many ways.

One considerable idea on the usage of ALOHA is not to let the graders know the grading scale but grade only with the points so that the graders do not have any idea nor effect on the forming of the final grade. This also allows the teacher to set the grade limits based on a certain wanted distribution. The downside of this is that it constricts the authority of the grader. Experienced graders may feel that their work is interfered if they do not get to decide about the grade themselves.

The tool is useful in courses where there are several graders but with only one grader the benefits are mostly limited to the more comfortable grading process. Of course if the grading takes several days of time the tool might help the grader to stay consistent throughout the whole process. Still, the construction of a grading template for just one person might require too much effort compared to its benefits.

The objectivity aspect of ALOHA is not that useful when an assignment is graded with the scale accepted/rejected (especially if almost every student should pass the assignment). The feature that calculates the actual grade is then obsolete but ALOHA can still be used to calculate possible bonus points. Nevertheless, the tool can be used to write good, consistent feedback texts.

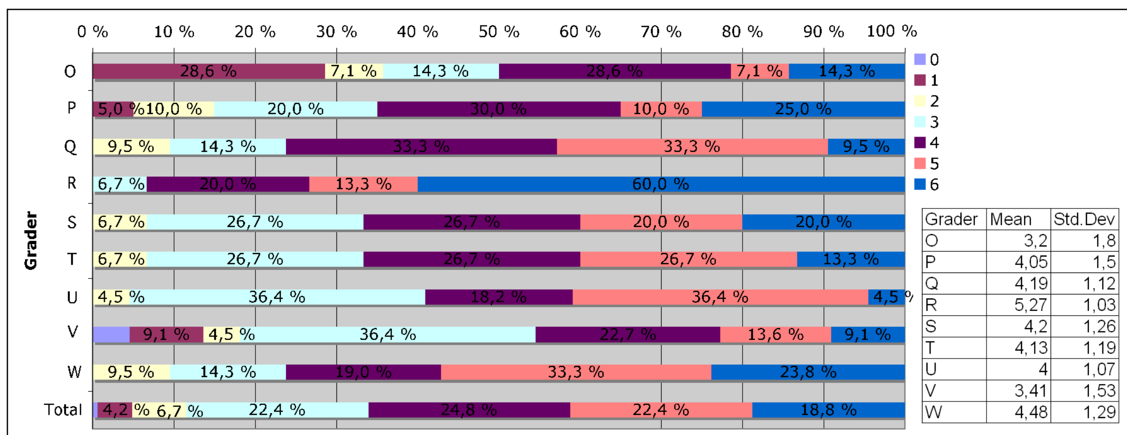
From the teachers point of view ALOHA is useful for two reasons: maintaining the objectivity and ensuring that each student will receive a personalized feedback email. From the graders perspective ALOHA is most useful because of facilitating the grading process with the phrase templates and customization features. We paid high attention on listening to the graders' opinions during the development process and accomplished a system that the graders are willing to use.

So far ALOHA is used only in programming courses, but its usage is actually not at all binded to programming nor even computer sciences. The limiting issue is that the building of a grading for an assignment requires moderate programming skills. Because it does not really have to be the teacher himself who does this, the tool could be used in other disciplines too for example to grade project works or essays.

6. CONCLUSIONS

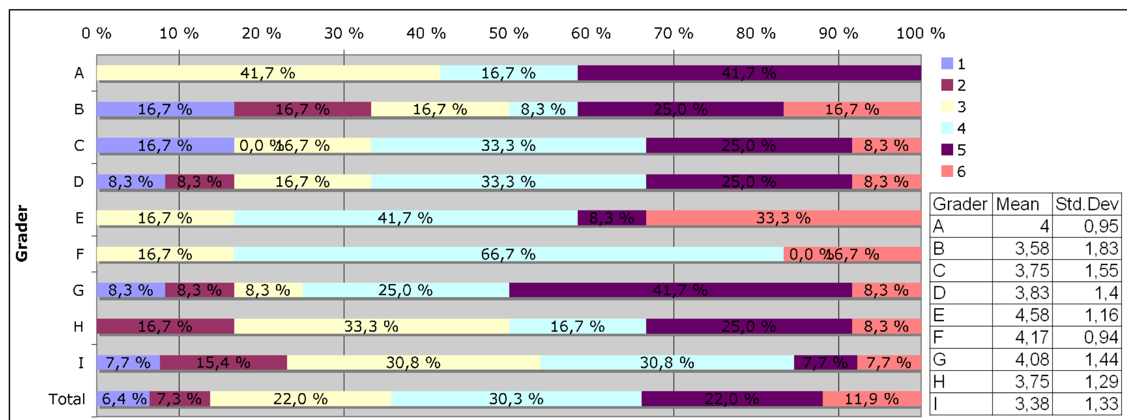
We have introduced few problems related to using multiple graders which is common in mass courses. ALOHA was built to facilitate these problems concerning consistency and objectivity in grading but also to make the grading process more comfortable for the grader. Despite many tools exist to support fully automatic assessment, we found ALOHA to be unique with its features in supporting manual assessment and especially the creation of written feedback.

The tool has been taken to use and it seems to make the grading process more convenient for the graders and also for the teacher. To test the objectivity we analyzed the grading



Grader	Mean	Std.Dev
O	3,2	1,8
P	4,05	1,5
Q	4,19	1,12
R	5,27	1,03
S	4,2	1,26
T	4,13	1,19
U	4	1,07
V	3,41	1,53
W	4,48	1,29

Figure 3: Distribution of grades by graders (O to W) using only traditional rubrics (quite similar to the ones in ALOHA)



Grader	Mean	Std.Dev
A	4	0,95
B	3,58	1,83
C	3,75	1,55
D	3,83	1,4
E	4,58	1,16
F	4,17	0,94
G	4,08	1,44
H	3,75	1,29
I	3,38	1,33

Figure 4: Distribution of grades by graders (A to I) using rubrics in ALOHA

distributions of the same course in two years statistically and the results indicated that ALOHA removes the problem of couple of graders who are clearly either stricter or more lenient than the others.

7. REFERENCES

[1] T. Ahoniemi and T. Reinikainen. ALOHA - a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, 2007.

[2] K. Ala-Mutka and H.-M. Järvinen. Assessment process for programming assignments. *Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT'04)*, 2004.

[3] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, June 2005.

[4] K. Becker. Grading programming assignments using rubrics. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 253–253, New York, NY, USA, 2003. ACM Press.

[5] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4, 2005.

[6] H. J. C. Ellis. Self-grading: an approach to supporting self-directed learning. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 349–349, New York, NY, USA, 2006. ACM Press.

[7] S. Habeshaw, G. Gibbs, and T. Habeshaw. *53 Problems With Large Classes*. Technical and Educational Services Ltd., Bristol, U.K., 1992.

[8] J. Hamer, C. Kell, and F. Spence. Peer assessment using Aropä. In *ACE '07: Proceedings of the ninth Australasian conference on Computing education*, pages 43–54, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[9] A. Trivedi, D. C. Kar, and H. Patterson-McNeill. Automatic assignment management and peer evaluation. *J. Comput. Small Coll.*, 18(4):30–37, 2003.

[10] T. Winters and T. Payne. Computer aided grading with Agar. In H. R. Arabnia, editor, *FECS*, pages 245–251. CSREA Press, 2006.

Publication (vii)

Ahoniemi, T., Karavirta, V. Analyzing the Use of a Rubric-Based Grading Tool *In: The Proceedings of ITiCSE 2009*, June 2009, Paris, France.

Analyzing the Use of a Rubric-Based Grading Tool

Tuukka Ahoniemi
Department of Software Sciences
Tampere University of Technology
Finland
tuukka.ahoniemi@tut.fi

Ville Karavirta
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
vkaravir@cs.hut.fi

ABSTRACT

Over the years, a lot of research has focused on how to assess programming courses. For programming courses, semi-automatic assessment combining automatic and manual feedback has been shown to be a good solution. In this paper, we will focus on the manual assessment part and analyze the use of a rubrics-based grading tool on larger courses with multiple graders. Our results show that the use of such tools can support objective grading with high-quality feedback with reasonable time usage. Finally, we will give some pointers for teachers intending to adopt such tools on their courses.

Categories and Subject Descriptors

K.2.1 [**Computing Milieux**]: COMPUTERS AND EDUCATION—*Computer and Information Science Education*

General Terms

Human Factors

Keywords

Grading; Rubrics; Programming; Assessment; Mass courses

1. INTRODUCTION

The authors, like many other Computer Science educationalists, have taken the long path of search for the best practices to give programming education, or more precisely, how to assess programming courses. The courses are often held as mass courses with hundreds of students. The search is thus an on-going struggle between doing as much as possible automatically, for instance, using Computer Aided Assessment (CAA) tools, and still retaining the personal touch for the students by providing formative assessment.

Formative assessment has been found to be critical for students' learning [6]. However, formative assessment is not widely used [7], mainly due to the increasing student/staff ratio, demands outside teaching (research, administration, etc), and increasing concern with attainment standards [11].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE '09 Paris, France
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

For programming courses, *Semi-Automatic Assessment* (SAA) has shown to be a good combination of easing the teacher's workload while still supporting the students well enough [3]. In SAA, the students' programming projects are tested with CAA tools for their correctness. Besides correctness, automatic assessment can also be extended to measure static parts, like coding style or algorithmic performance. After the program is functioning "well enough", a teaching assistant (or the teacher him/herself) reads the code, now focusing on other things like program design and individual implementational Tips&Tricks. As a result, the student receives personal feedback but the grader has not had to waste time on testing the program. So, Semi-Automatic Assessment is a combination of CAA and manual grading.

Multiple good CAA tools exist (see [8]) for the aid of the automatic part but this paper is about facilitating the other part: the manual grading. To put it short, the following problems have been the basis for the original work, more widely presented in [1]:

- Consistency: The mass courses often hold not only many students, but many graders. How to obtain consistency amongst them? The grading should not be affected by who is the grader.
- Objectivity: All student submission should be graded with the same criteria and each criterium proven to be taken into account. So, the grader shouldn't just read the student submission through and in his mind decide the grade with "his gut feeling".
- Writing good feedback over and over again: Especially in elementary programming courses, the submissions – and thus the feedback written about them – resemble each other. This can be facilitated to make the grader more efficient and the feedback better.

Habeshaw et al. [9] argue that to ensure consistency among graders, they need to be properly trained and required to use marking schemes to direct their attention to the appropriate things. Becker suggests a similar approach to be used for programming assignments, and also defines the schemes as rubrics [5]. The idea of rubrics is to divide the grading into small enough parts so that each part can be objectively graded following given instructions.

The use of computer-assisted grading rubrics has been shown to help in solving the first two problems [1] and also to improve the speed of assessment [4]. To make the use of rubrics more convenient as well as combining the writing of the feedback to them, we have previously developed a grading tool called ALOHA [2].

After two full years of the use of ALOHA tool, it is time to evaluate how it has served its original purpose in helping with the three previously mentioned problems. However, the ALOHA tool itself serves here only as an example. What this paper is about is what the title says: *Analyzing the Use of a Rubric-Based Grading Tool.*

In this paper, we introduce a few relevant measurements we have performed for a set of gradings done with ALOHA. What is measured is not the students, but the behaviour of the teaching assistants (TA) who used the tool. Each measurement has a certain purpose resulting into a discussion on the different ways of using the tool. As a general result of this paper, we will provide results on the effectiveness of the tool as well as instructions on what to take into account when taking such a tool into use.

2. ALOHA IN BRIEF

To get the idea of what was the concrete use case in this study, we briefly introduce the main idea of the tool we used: ALOHA. The tool with all its features is explained in more detail in previous work [2].

The purpose of ALOHA is to provide grading rubrics online in a way that each grader has to fill them correctly and in a coherent way. ALOHA also helps the grader in creating feedback during the grading process. For the teacher, the tool is also a useful tool for general management of the submissions and the feedback process (e.g. assigning submissions for graders, sending feedback mails, and collecting grades).

After logging into the tool, the grader is presented with a list of student submissions the teacher has assigned to him. The grader chooses one of the submissions and is taken into the grading view, shown in Figure 1.

The grading view represents the grading rubric. The grading is based on a categorized hierarchy shown in the left of Figure 1. As the grader is going through the hierarchy, he selects the suitable grade for each small grading item of which the whole grade is eventually formed. At the same time, the grader will write the feedback for that part of the grading either manually or using predefined phrase templates. These templates are created by person responsible for the rubric (typically, the teacher).

The phrase templates form one of the main features in ALOHA called *the semi-automatic phrasing.* This means that the grader can choose suitable phrases to be added directly to the feedback text. When a phrase is selected, it is copied as text into one of the Positive/Negative/Neutral -feedback forms, depending on the type of the phrase (each phrase is classified as one). The text in the form is editable so the grader may modify it if he wishes. Often the idea is that the phrase the grader chooses from the list is purposefully incomplete and the grader manually completes the phrase, thus personalizing the feedback. After the whole submission is graded, the grader needs to accept the complete grade and the generated feedback text with the possibility to yet modify it, as a whole.

3. MEASUREMENTS

In this section, we present all the measurements we conducted and what we were hoping to see from their results. The results itself are also combined to this section.

3.1 Experiment Settings

The use scenario here is the grading process of the second elementary programming course in Tampere University of Technology (CS1 level) held in spring 2008. In the course, the students create individually a larger programming project. The grading is done semi-automatically so that the students submit their work to a CAA system, which checks the functionality and static coding style measurements. After reaching sufficient level, each submission is manually read and assessed by a teaching assistant using ALOHA.

The manual assessment process is profound and should take from 30 minutes up to two hours of the teaching assistant's time. Each of the nine TAs had 16 to 18 submissions to grade resulting into altogether 150 student submissions. As a result of the grading, all the students were supposed to receive a well-written, personalized feedback mail and a grade between 0 to 6, 1 meaning passed and 6 being the best possible grade (quite rare).

To monitor the use and TAs' grading behaviour, we made ALOHA to log every action that was done. The whole data we analyzed then consists of a vast amount of timestamped actions, grades, and feedback mails. Of those we measured the following: objectivity between different graders, different time usage measurements, and the (statistical) quality of the feedback. To reinforce our findings we also performed statistical analyses for the results.

3.2 Objectivity in Grading

As explained in Section 1, one of the problems with multiple graders is objectivity: the grade should not be affected by who is the one assessing it. Rubrics are designed to solve this problem. In the first evaluation of ALOHA, the effect on achieving this was measured and successfully found to be positive [1]: In this same course, before using ALOHA there was a statistically significant difference between the grades given by a couple of the TAs. After introducing ALOHA the next year, there were no longer such statistically significant differences.

To confirm these results from previous years, we replicated this measurement. We performed a one-way anova test for the mean grades given by different TAs. We wanted to see if there were the kind of differences that someone would generally give better grades than others or vice versa. The results are presented in Table 1 combined with the previous results for comparison.

The result of this measurement remained positive as this year showed no statistically significant differences between the graders ($p=0.1561$). This means that now with measured use of two years, a rubric based tool has a positive effect on achieving objectivity.

3.3 Time Spent Using the Tool

As we wanted to know more about how the teaching assistants use the tool, we analyzed the time spent on various tasks in the grading process. The tasks we were interested in were the time spent on grading a single grading page (such as the one shown in Figure 1) and the time spent on modifying the final feedback. In addition, the total time of these will be examined. The results of this analysis are presented in Table 2 (outliers have been removed from the data). However, it should be noted that these times are not equal to the actual time spent on grading the works, but it does give

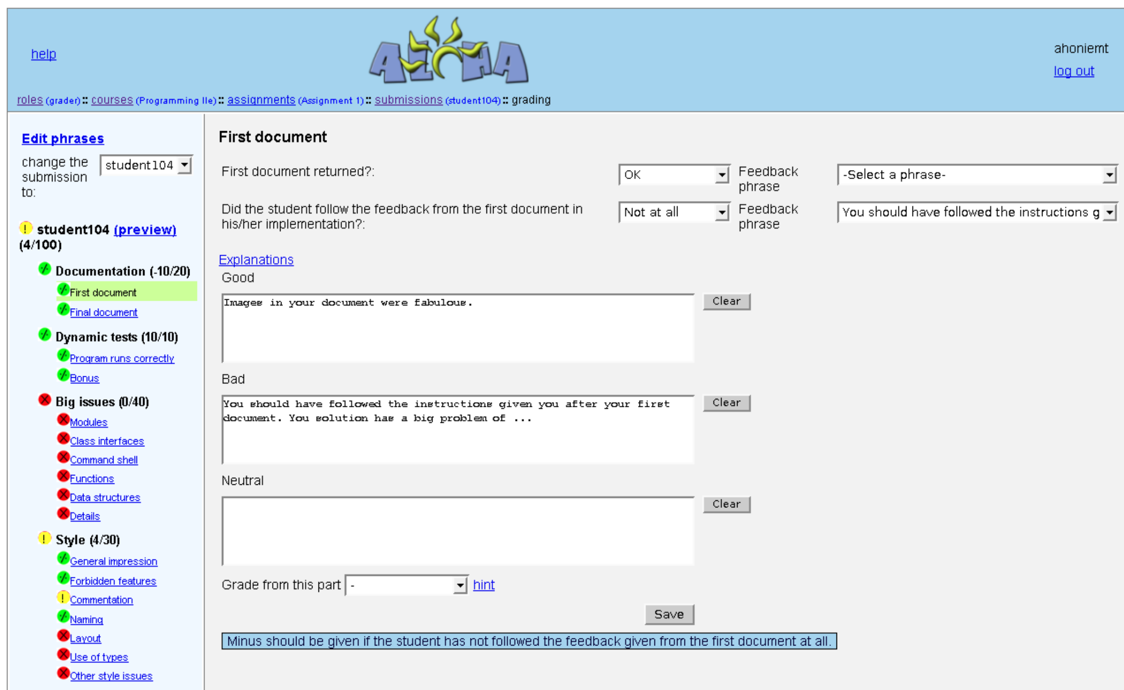


Figure 1: Screenshot of the grading view in ALOHA

Table 1: Means and standard deviations of grades given by graders in three years (Y1-Y3). Scale is 0-6. Years 2 & 3 used ALOHA. Statistically significant difference marked with *. The TAs were different in each year.

Y1		Y2		Y3	
Mean	St.d	Mean	St.d	Mean	St.d
*5.3	1.0	4.6	1.2	4.6	1.4
4.5	1.3	4.2	0.9	4.5	1.1
4.2	1.3	4.1	1.4	4.3	1.2
4.2	1.1	4.0	1.0	4.1	1.2
4.1	1.2	3.8	1.4	4.0	1.0
4.1	1.5	3.8	1.6	3.9	1.6
4.0	1.1	3.8	1.3	3.6	1.7
*3.4	1.5	3.6	1.8	3.6	1.8
*3.2	1.8	3.4	1.3	3.5	1.5

us a rough minimum estimate of it. Besides the rough time estimate these results also give us an examination on how differently the tool is actually used.

The measurement for the time it takes TAs to fill a single grading page of the rubric is the time between the page being opened and the feedback being saved. As can be seen from the table, there were quite significant differences in the time spent on a grading page (in fact, there were even statistically significant differences). However, there was no correlation between the time spent on grading and the grades that were given.

One interesting thing we observed was how the time for grading page changes while the TA grades more assignments. The time for category grading decreased for all but one (TA6). This suggests that the tool is useful, since it makes grading faster with experience. However, the time change was not too radical for us to be concerned about the quality of the grading.

Another timing statistics we measured was the time used to customize the final feedback that is sent to students via email. Here, there were small differences between the TAs. Some assistants did little customization at this point, which can suggest that the ALOHA tool worked well while grading the categories. On the other hand, we feel that the TA should at least read through the final feedback and make sure it is formatted correctly. Thus, we feel that there is room for improvement in the tool. Currently, the assistant has to click on a link to go and modify the final feedback. This should be improved to include the feedback modification functionality on the final acceptance page of the grading.

The total times for grading one assignment are the sum of the total time used for assessing all the grading pages of one assignment and the time used to modify the final feedback. The total times are, of course, varying radically. This, however, is quite expected and nothing to be too alarmed about since some of the graders might use the tool only at the end of the grading after reading the code, and some use it parallel. In fact, informal discussions with the TAs confirm these different behavior patterns. More detailed inspection of different behaviors cannot be done through analyzing usage logs, but would require studying their actual work.

3.4 Days on Task

To get some sense on when the assistants use the system, we examined the number and distribution of the days they used ALOHA for *grading*. These are displayed in Table 3 (we have left out days after the initial grading deadline, since there were only few extended deadlines and resubmissions). The most positive notion we can make is that most of the TAs started the assessment earlier than the last few days before the assessment deadline (on the beginning of the last week shown). In addition, they worked on several, often consecutive days. Only TA4 labored through all the assignments in just three days, two days being right before the deadline. This kind of behaviour is human, but it should

Table 2: Statistics of the time used for a grading page, final feedback, and the total time for one assignment.

Teaching Assistant	Mean time for grading page (s)	Mean time for final feedback (s)	Mean of total time for assignment (min:sec)
TA1	137	51	38:14
TA2	144	236	41:14
TA3	100	268	43:58
TA4	71	471	24:05
TA5	76	51	23:37
TA6	43	160	17:36
TA7	58	41	15:02
TA8	60	81	16:27
TA9	74	95	22:57

be discouraged by the course personnel due to the inevitable grader fatigue that can affect grading.

3.5 Quality of Feedback

The overall quality of the feedback is a subjective measurement and would basically require qualitative analysis. A couple of quantitative measurements can, however, be done to get some indisputable measurements of the feedback which are affecting the overall quality.

3.5.1 Amount of Lines

The overall length of one feedback mail can be used as one quantitative measurement for the quality of the feedback. Longer feedback of course does not mean a better feedback, but a good feedback requires at least a proper amount of text. Also, if the feedback mail is long, the student feels that the assistant has used time and effort for the grading and really wants to help the student. This is an important aspect of how the students value the given assessment.

We measured the amount of lines in each feedback mail. The average length of all the feedback mails was as high as 113 lines (multiple pages of text). So, it can easily be concluded that in general the feedback was of high quality concerning this measurement. There were differences between the graders in this category. Some of the assistants wrote noticeably longer feedback than others, even as long as 269 lines! However, even the shortest feedback mails were a bit over 60 lines, which can still be found as a proper amount of feedback (especially if the submission had nothing wrong).

3.5.2 Percentage of Predefined Phrases

To get insights on how much the assistants customize the feedback for students, we measured the amount of personalized feedback. A typical approach to this would be to assess it qualitatively, introducing some subjectivity. We took a completely different approach, though.

As mentioned, ALOHA includes the possibility to use predetermined phrases for the feedback, the *the semi-automatic phrasing*. Similarity of a document consisting of only the predetermined phrases with the amount of personalized feedback can be measured by using a plagiarism detection system. We used a system called Nalkki [10], developed at the Tampere University of Technology. A feedback with minimum personalization will then score high against the "all-phrases-document" on the plagiarism measure and a personalized feedback will have a lower score. However, this again is not directly correlating with quality as the phrases itself should be of high quality too. A high use of them might indicate that the grader is using the tool more efficiently. On the other hand, a too high value tells that the feedback is not really personalized at all and the student might spot that, especially if comparing the feedback with peers.

The overall percentage of predefined phrases in all feedbacks was a bit over 50%. Here, we also observed great differences between the graders: some of the assistants had an average percentage of over 70% where as couple had used predefined phrases (without any modifications) only for around 30%. We find this range of values quite acceptable and understandable, but as one of the assistants had some feedbacks with the value as high as 90%, there might be more than just efficient use of the tool. This part requires more guidance, control and most easily more phrases that really require manual additions.

A positive observation was that there was no correlation at all with the length of the feedback and the amount of predefined phrases. On the basis of these numerical results this means that a long feedback can also be written with the efficient use of the semi-automatic phrasing feature.

4. DISCUSSION AND CONCLUSIONS

So, how can the data collected and the results of the analysis be used? The separate measurements themselves provide interesting statistics from the different ways of using the tool as well as proof of the effectiveness of the tool. The results indicate that different graders have indeed different working methods. Some prefer reading the submission first as a whole and some read it parallel to writing the feedback and grading it. This is rather obvious, but despite these different habits, our results confirm the following:

- All the graders used well enough *time* with the tool. None of the teaching assistants used too much time with the tool. In addition, the time required for grading one assignment decreased as the number of graded assignments increased.
- The TAs wrote *proper feedback* combining the provided phrases and personalized feedback.
- The grading remained *objective* – the previously existing problem of non-objective grading amongst multiple TAs seems to be extinct.

Together these results show that, in general, a rubric-based tool with convenience features (here, semi-automatic phrasing) strongly supports the manual grading process. Thus, the original goal and purpose of the tool – objective grading with high-quality feedback in reasonable time – is met.

Table 3: Days used for grading the assignments. X marks a day the TA graded assignments. The dates mark the first day of a week, starting from Mondays (day/month).

TA	26/5	2/6	9/6	16/6	Days
TA1		XXX X	XXX	XX	9
TA2		X X	XXXXXX	X	9
TA3			XX XXX	X	6
TA4	X		XX		3
TA5		X	XXXXX X	XX	9
TA6	XXXXX				5
TA7		XX X	XXXX X	X	9
TA8	X	X X	XXX	X	7
TA9	XXXXX X	XXXX			10

Based on the benefits indicated by our results, we firmly suggest the use of this kind of tool for the teachers responsible for courses with multiple graders. Besides aiding in grading management, it aids in ensuring objectivity and a proper level of feedback. However, creating a good grading template which includes the rubric and the phrases requires effort. Once it's done for one course, it can be reused between different years with only small modifications.

We also received good experience on collecting data from the use of a grading tool. When designing a new grading tool it is also a good idea to integrate data collection to the tool itself, providing immediate statistics for the teacher on the grading process. One should also consider letting the graders themselves access the statistics to monitor their own work compared to other graders. In the future, we intend to do further research into what the TAs' grading process is like. However, if the monitoring is too detailed and the data is not interpreted discretely, there is the risk of the graders feeling a lack of trust and that their work is being monitored too much.

While our results provide sufficient insurance on everything running as hoped, one must remember that the teaching assistants in focus were well trained both for the tool and for the general guidelines for the grading. Thus, we want to conclude with a few important points for teachers intending to use rubrics-based grading tool to help them acquire the benefits reported in this paper:

- The rubric and the phrases for semi-automatic phrasing need to be designed carefully. For large parts, the given feedback is based on the phrases in the rubric.
- The graders need to be well instructed to know how they are expected to use the tool, what sort of feedback they are expected to write, and how to assess.
- A rubric-based grading tool can be a really good aid for the whole course staff, but it does not replace the independent responsibility and skills of the graders.

5. ACKNOWLEDGMENTS

Tuukka Ahoniemi is funded by Digia Plc and partly by the Foundation of Nokia.

6. REFERENCES

[1] T. Ahoniemi, E. Lahtinen, and T. Reinikainen. Improving pedagogical feedback and objective grading. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 72–76, New York, NY, USA, 2008. ACM.

[2] T. Ahoniemi and T. Reinikainen. Aloha - a grading tool for semi-automatic assessment of mass programming courses. In *Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research*, pages 139–140, New York, NY, USA, 2006. ACM.

[3] K. Ala-Mutka and H.-M. Järvinen. Assessment process for programming assignments. *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*, pages 181–185, 30 Aug.-1 Sept. 2004.

[4] L. Anglin, K. Anglin, P. L. Schumann, and J. A. Kaliski. Improving the efficiency and effectiveness of grading through the use of computer-assisted grading rubrics. *Decision Sciences The Journal of Innovative Education*, 6(1):51–73, January 2008.

[5] K. Becker. Grading programming assignments using rubrics. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 253–253, New York, NY, USA, 2003. ACM.

[6] P. Black and D. Wiliam. Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, 5(1):7–74, 1998.

[7] F. J. R. C. Dochy and L. McDowell. Assessment as a tool for learning. *Studies In Educational Evaluation*, 23(4):279–298, 1997.

[8] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4, 2005.

[9] S. Habeshaw, G. Gibbs, and T. Habeshaw. *53 Problems With Large Classes.* Technical and Educational Services Ltd., Bristol, U.K., 1992.

[10] P. Sirkkala and S. Puonti. Nalkki-project - tool for plagiarism detection using the web. In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 229–230, Koli National Park, Finland, 2007. ACS.

[11] M. Yorke. Formative assessment in higher education: moves towards theory and the enhancement of pedagogic practice. *Higher Education*, 45(4):477–501, 2003.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3526-0
ISSN 1459-2045