Koli Calling 2009

8th International Conference on Computing Education Research

Arnold Pears and Lauri Malmi

Department of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden

Technical report 2009-004
February 2009
ISSN 1404-3203

From the Conference Chairs

This volume collects together the papers presented and discussed at the 2008 Koli Calling International Conference on Computing Education Research.

This volume is the culmination of more than a year of planning and effort on the part of both the local organising committee and the conference chairs. However, we were not working alone. Without an active community of researchers doing quality research and writing papers, a conference like Koli has no function or purpose. Consequently, a large part of the success of Koli Calling lies in its vibrant research community. It is your submissions that have made it possible for us to select this year's crop of interesting and thought provoking contributions.

During the preparations for the 2008 conference we embarked on a process of clarification and innovation. The major outcomes of that process are a more well defined submissions and review process based around the use of EasyChair. We have also crafted new guidelines for the evaluation of the conference submission categories; which we hope are useful to both authors and reviewers alike. We also introduced the Tools Workshop submission category and the Tool Award in the 2008 call for contributions. We wish to extend our sincere thanks to Ari Korhonen who was Tools Workshop Chair for the 2008 conference. A new role, and one that he managed with panache.

So now, without further ado, we leave you to the further perusal of the contents of the volume, in the hope that you will find its content both elucidatory and inspirational.

Lauri Malmi and Arnold Pears
Koli 2008 Conference Chairs

Table of Contents

From the Conference Chairs.

Conference Keynote.

Educational Research and Design of the Virtual Learning Environment *(invited talk)* 1
Erik deGraaff

Research Papers.

The Same But Different – Students's Understandings of Primitive and Object Variables 5
Juha Sorva

Diagnosing Learners's Problem Solving Strategies Using Learning Environments with Algorithmic Problems in Secondary Education 16
Ulrich Kiesmüller

Understanding TDD in Academic Environment: Experiences from Two Experiments 25
Sami Kollanus, Ville Isomöttönen

Why Using Robots to Teach Computer Science can be Successful Theoretical Reflection to Andragogy and Minimalism 32
Marja-Ilona Koski, Jaakko Kurhila, Tomi A. Pasanen

A Tablet PC Capture Platform for Explanograms 41
Tony Clear, Jacqueline Whalley, Jonathan Hill, Yong Liu, Arnold Pears, Beryl Plimmer

Implementing a Contextualized IT Curriculum: Ambitions and Ambiguities 51
Matti Tedre, Erkki Sutinen, Fredrick Ngumbuke, Nicholas Bangu

A Typology of CS Students's Preconditions for Learning 62
Maria Knobelsdorf

Discussion Papers.

Understanding Computing Stereotypes with Self-Categorization Theory . . 72
Michael Hewner, Maria Knobelsdorf

Helping Students Debug Concurrent Programs 76
Jan Lönnberg, Lauri Malmi, Anders Berglund

Minority Report - Computer Science Skills Perceived by Students in Different Disciplines 80
Tuukka Ahoniemi

Student-Generated Podcasts for Learning and Assessment 84
Colin Johnson

Algorithm Recognition by Static Analysis and Its Application in Students' Submissions Assessment 88
Ahmad Taherkhani, Lauri Malmi, Ari Korhonen

Students' Individual Differences in Using Program Visualizations 92
Essi Lahtinen

Tools Session.

PatternCoder: A Programming Support Tool for Learning Binary Class Associations and Design Patterns 96
James Paterson, John Haddow, Ka Fai Cheng

Automatic Assessment of Program Visualization Exercises 101
Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, Tapio Salakoski

JLS/JLSCircuitTester: A Comprehensive Logic Design and Simulation Tool 105
David Poplawski, Zachary Kurmas

PeerWise 109
Paul Denny, John Hamer, Andrew Luxton-Reilly, Helen Purchase

Poster Presentations.

Towards Students' Motivation and Interest - Teaching Tips for Applying Creativity 113
Ralf Romeike

Presentation of Automatic Conflictive Animations 115
Andrés Moreno, Niko Myller

Is Automatic Evaluation Useful for the Maturity Programming Exam? . . . 117
Bronius Skupas, Valentina Dagiene

How a Contextualized Curriculum work in Practice 119
Joseph Longino, Mikko Vesisenaho

Educational Research and Design of the Virtual Learning Environment

Erik de Graaff
Delft University of Technology
Netherlands

E.deGraaff@tudelft.nl

ABSTRACT

The aim of higher education is to enable students to acquire knowledge and to exercise cognitive skills in order support them in their preparation for a professional career. Rather than transferring knowledge in face-to-face contact the modern teacher has to design a stimulating learning environment. The success of educational models, like Problem-Based-Learning and Active Learning is often explained by the motivating effect of discussing real-life problems in small groups of students. The technology of virtual reality provides new possibilities to involve students in learning activities. No longer do groups of students (and their teacher) have to meet at a fixed time and place. Simulations and gaming can motivate students to engage in activities that make them learn. The biggest challenge for the teacher is to imagine what is motivating for a present day student.

Categories and Subject Descriptors

K.3.2 Computer and Information Science Education

General Terms

Management, Documentation, Human Factors.

Keywords

Student, Learning, Environment, Engineering Education Research.

1.INTRODUCTION

Teaching is a profession with a long and respected history. If you aim to prepare your children for a lifetime career, you want them to train with the best. The traditional conception of teaching is almost identical to the transfer of knowledge in face-to-face contact. The teacher provides information on topics, which are novel to the learner and explains how to apply this knowledge. Consistent with the image of content expertise, a traditional teacher supervises assignments to practice the relevant skills and judges the student's achievements.

Teaching and learning are often taken to be complementary: the students learn what the teacher teaches. In many instances, however, this is obviously not the case. A major complaint of teachers all over the world is that the students are unable to reproduce what they have been told. Learning is an activity in its origin. The etymological roots of the word ´learning´ go back to the activity of finding a track [10]). Similarly, the meaning of the word "teaching" is derived from roots referring to the act of pointing at something or pointing something out as is still clearly evident in many European languages:

Dutch: *onderwijzen*

German *unterwissen*

Scandinavian languages *unterwisen*

French: *ensigner*

The English word teaching goes back to a Saxon root ´tecam´, with a similar connotation. Hence, since ancient times, a teacher is someone who transfers certain knowledge or skills to a learner by pointing it out. The teacher explains and demonstrates, enabling the learner to follow in his footsteps. Transfer of knowledge is part of the job of a teacher, but it is by no means the most essential aspect. The intensive personal contacts between teacher and student allow for the expansion of the task of the teacher to include the teaching of moral values and the formation of personality. In a traditional context student's motivation is not much of an issue. Students are expected to attend classes for their own good, and if they are not yet capable of appreciating their good fortune, they are simply obliged to go. The authority, or the capability of the teacher to maintain order in the classroom is one of the most important traditional didactic skills.

The emergence of the knowledge society entails fundamental changes in the processes of teaching and learning. On the one hand there is the ever-increasing number of students that need to be served, turning teaching into some sort of mass production. A negative image of the future of higher education is that of immense learning factories, where teachers act as drill masters, each instructing the students within the limits of a narrow specialty. In Holland, like in other West European countries, this development takes shape as a wave of mergers between institutes of Higher Education. On the other hand there are the opportunities for enhancement of learning provided by technological innovations. Knowledge is just a click away and interactive software for practicing is easily available. The creative challenge facing the modern teacher is to collaborate in the design a learning environment that stimulates student's self-directed learning processes.

2.MOTIVATION FOR LEARNING

A stimulating learning environment is one that captures and retains the attention of the students. The competition is stiff. Young people today are swamped with incentives competing for

Koli Calling '08, November 13-16, 2008, Koli, Finland. Copyright 2008 ACM 978-1-60558-385-3/08/11.$5.00.

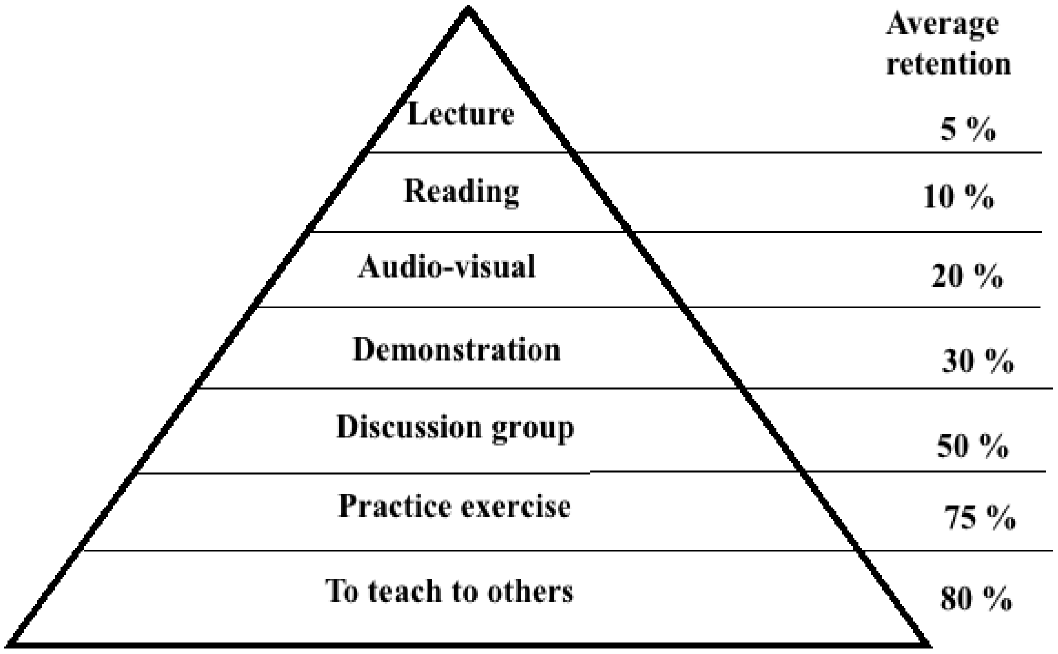


Figure 1: The `Pyramid of Bales` (after: Van der Vleuten 1997)

their attention. In order to be effective teachers will have to understand the basic mechanisms of human motivation.

Attraction and motivation are key areas of psychological research, focussing on the forces that drive our behaviour. A well-known example is the fact that people's pupils get bigger when they look at something they like. Consequently, when someone with dilated pupils is looking at you that is pleasing, because it gives you the impression this person finds you attractive. Even in antiquity women were aware of this fact and they applied the drug belladonna to create this impression. In modern times advertisement designers use this information as part of a selling strategy.

The concept of motivation brakes down in two components: the intensity and the direction of behaviour (what you want to achieve and how badly you want it). A fundamental distinction is the one between intrinsic motivation, coming from inside the person and extrinsic motivation, when the person is driven by external stimuli like rewards or punishment. Psychological research has demonstrated repeatedly that intrinsic motivators are in particular strong in determining the direction and the short time intensity of behaviour (doing something that is attractive to the person, or that the person perceives as immediately useful). However, in many instances this does not result in enduring behaviour. After a while the attraction fades, or other more attractive alternatives come up. Consequently, the person governed by intrinsic motivation changes direction easily [1].

In the design of a learning environment the limitations of intrinsic motivation need to be countered. Take for instance the successful educational concept Problem Based-Learning.

One of the key features of this method is the appeal on intrinsic motivation by real life problem situations [7]. Therefore, PBL tutors must be trained to ensure that the students keep on the right track. Collaborative learning, another key feature of PBL, is also an important aspect in explaining student's efforts [11].

Educational research shows that learning is more efficient when students are actively involved [3,8]. One-way transfer of knowledge in lectures results in poor learning. Van der Vleuten [13] quotes the USA researcher Bales who depicts the relationship between active involvement of the students and retention of relevant knowledge in the form of a pyramid (See Figure 1.)

Evidently, an effective learning environment should stimulate students to find information for themselves, rather than to have them receive passively pre-processed pieces of information. Of course, it depends on the particular situation how best to engage the students. Nevertheless, educational concepts like 'discovery learning', 'learning-by-doing'. 'experiential learning' and 'student centred learning' suggest exploiting human traits like curiosity,

challenge and self-determination [9].

3.LEARNING IN VIRTUAL REALITY

Without a doubt Computer and TV screens are great at capturing attention. The term addiction is even used regularly. Both in the passive sense of watching TV, as well as in interactive online games a lot of time is spent behind the screen. Naturally, educational designers would like to emulate this effect. However, we should no forget there are some major differences between school and the world of fantasy. For one thing, the production of successful movies and games rests on the efforts of many people sustained by budgets bigger than educational designers can dream of. But also, the difference in purpose can be difficult to surmount.

Games and movies are made for amusement. They appeal to a largely unconscious human wish to escape reality. The relationship between fantasy and reality is a complicated one. In order to explain the relationships Fred Alan Wolf [14] elaborates the concepts of knowledge and observation used by the ancient philosopher Plato. In his well-known allegory of the cave Plato suggests man cannot observe reality directly. The only thing we

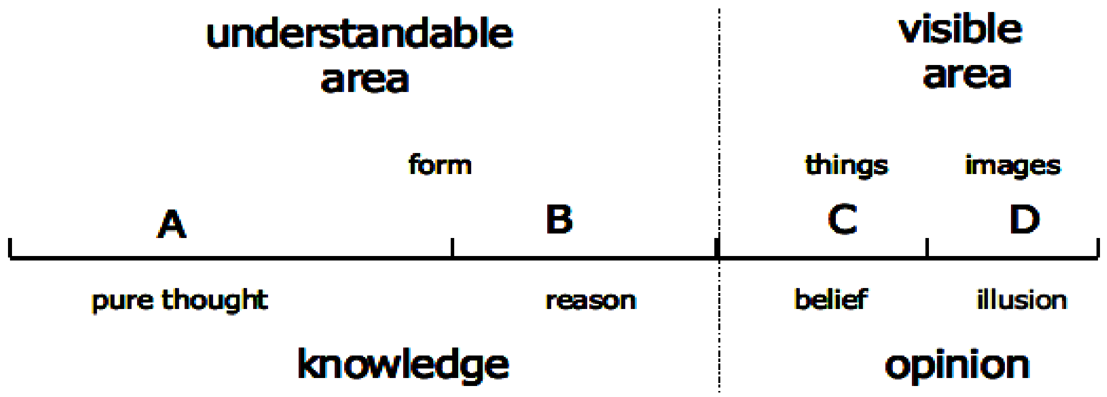


Figure 2: Plato's line of knowledge

see is shadows cast on a wall and that is what we take to be real. Plato uses a line divided into segments to depict the different aspects of human knowledge about the world. The entire line A+B+C+D represents everything we can observe or think about the world. Horizontally, the line divides the objects of our knowledge in things that can be observed in the outside world above the line and below the line the world of thoughts, pure thought, not contaminated by observations, and reason on the left and on the right beliefs and illusions. In the reality below the line "Knowledge" is more important than "Opinion", as expressed by the length of the line segments. According to Plato there is a fixed ratio between the line segments. A is longer than B and C is longer than D. In this constellation the ratio's (A+B) / (B+C+D); A/(A+B); C/(C+D) are constant (See Figure 2.).

Wolf uses the graph of Plato's line of knowledge to demonstrate the changes in perception of reality in our times (See Figure 3.). Today, opinions take precedence over factual knowledge. The image has become more important than the object it represents. On the inverted line, the world of illusions has become the most important part. Shadows are more important than physical things and reason is more important than pure thought. This image neatly explains a number of phenomena characteristic of the present era, for instance how actors can reach to a high position in politics. Or, why ex-politicians are warmly welcomed by large companies: "because they have experience in playing a role of command", as it was explained in a newspaper.

The philosophy behind Plato's line of knowledge fits in nicely with modern Constructivist educational theories. According to these theories our image of the world is not based on a fixed template, but rather re-constructed by the individual observer [10]. The consequences of the process of cultural change embodied by the inverted line of Plato merits serious considerations and could inspire much research.

4.CONCLUSIONS

Learning is a complex, many faceted, activity. The task of the teacher is to engage the students in learning activities. To capture the attention of the learners teachers have many different methods and techniques at their disposal. For instance, many teachers have been known to practice methods from the theater in their lecturing. Modern technology offers a plethora of opportunities to create stimulating learning environments based on computer simulations and gaming. Virtual reality offers endless opportunities to practice.

However, not all gaming is useful. Most of the time that people spent on games and simulations is aimed at distraction, or escaping reality. Trying to make the game contribute to learning could just take away the attractiveness. An emerging problem is that reality and fiction are getting completely mixed up. Emotions run higher with fictional stories than with true events and when a real disaster occurs, we use examples from fiction to underscore the extent of our horror. Take for example the 9-11-2001 destruction of the WTC in New York as compared to images from the Hollywood movie Armageddon. Today's consumers buy things, because it authenticates the image they like to project on the world. You become what you can imagine. As a consequence, learning how to shape your own image and how to get other people to accept that image becomes an important learning goal for today's schools.

Presently, game designers just follow their hunches. For instance, a lot of effort is spent trying to create games as real-like as possible. However, that may not always be true. Experiments with a driving simulator at the department of mechanical Engineering of TU Delft show that not all additional features aimed at increasing the authenticity of the driving experience, actually result in improved learning. More research is necessary to determine the changed relationship between fiction and reality more precisely. Some pressing research questions are:

- What are the factors that motivate people to engage in online games like Second Life?
- How do people build their avatar and how does the avatar relate to the person?
- If the avatar has achieved something, how does that affect the person in real life?
- What is it that people get in return for the time they spent in online communities?
- What is the transfer of learning experiences in virtual reality to the real world?

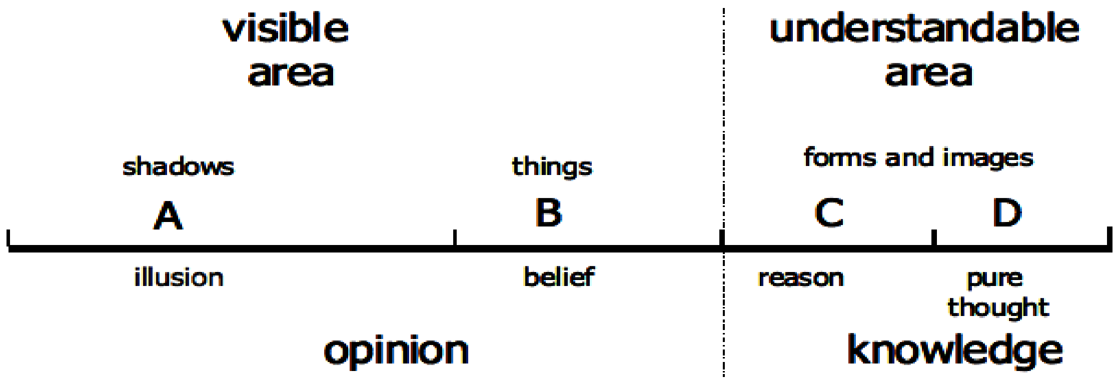


Figure 3: Plato's inverted line of knowledge

5.REFERENCES

[1] Aizen, I. & Fishbein, M. (1980) Understanding attitude and predicting behaviour. Englewood Cliffs: Prentice-Hall Inc

[2] Bales, R. F. (1992). National Training Laboratories, Bethel, Maine, USA

[3] Biggs, John B. (1999) Teaching for Quality Learning at University. Buckingham: Society for Research into Higher Education & Open University Press.

[4] Boud, D. & Feletti, G. 1991 The Challenge of Problem-based Learning. London: Kogan Page.

[5] Boud, D. & Miller, N (1996) Working with Experience; animated learning. London, New York: Routledge.

[6] Entwistle, N. (1993) Influences of the Learning Environment on the Quality of Learning. In: Th. Joosten, G. Heijnen & A. J. Heevel (red.) Do-ability of Curricula. Lisse: Swets & Zeitlinger, pp. 69-87.

[7] Graaff, Erik de, Fruchter, R.& Kolmos, Anette (2003) Problem Based Learning in Engineering Education (eds.) Vol.19. Theme issue of the International Journal of Engineering Education.

[8] Graaff, Erik de, Gillian N. Saunders-Smits & Michael R. Nieweg (2005) Research and Practice of Active Learning in Engineering Education. Amsterdam: Pallas Publications.

[9] Graaff, Erik de & Anette Kolmos (2007) Management of Change; Implementation of Problem-Based and Project-Based Learning in Engineering. Rotterdam / Taipei: Sense Publishers.

[10] Gijbels, D. , G. van de Watering, F. Dochy & P. van den Bossche ((2006) New Learning Environments and Constructivism: The Students' Perspective. Instructional Science. Vol. 34. No. 3. P. 213-226

[11] Jones, Ann & Kim Issroffb (2005) Learning technologies: Affective and social issues in computer-supported collaborative learning. Computers & Education. Vol. 44, Issue 4, P. 395-408.

[12] Skeat, W.W. (1993) The Concise Dictionary of English Etymology. Hertfordshire: Wordsworth editions Ltd.

[13] Van der Vleuten, C. P.M. (1997) De intuïtie voorbij [Byond Intuition] Tijdschrift voor Hoger Onderwijs, 15.1. p. 34-46.

[14] Wolf, Fred Alan (1996) The spiritual Universe. New York: Simon & Schuster.

The Same But Different

Students' Understandings of Primitive and Object Variables

Juha Sorva
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland
jsorva@cs.hut.fi

ABSTRACT

From qualitative analysis of student interviews emerged three sets of categories, or *outcome spaces*, describing introductory students' understandings of variables. One outcome space describes different ways of understanding primitive variables. Another describes different understandings of object variables. The third outcome space describes the relationship between the primitive and object variables, again from the point of view of the student cohort. The results show that learners create various kinds of mental models of programming concepts, and that the concept of variable, which is fundamental to most types of programming, is understood in various non-viable ways. With the help of the outcome spaces, teaching materials and tools can be developed to explicitly address potential pitfalls and highlight educationally critical aspects of variables to students. A software tool, which would engage students to interact with and manipulate a visualization of a notional machine, suggests itself as an intriguing avenue for future work.

Keywords: variables, references, students' understandings, misconceptions, phenomenography, CS1

1. INTRODUCTION

1.1 Research Question

The research presented in this paper is part of a project that explores introductory programming students' understandings of program execution and the notional machine. The research question relevant to this paper is:

> *In what different ways do introductory programming students understand the concept of variable?*

Both correct and incorrect understandings are of interest. I take a qualitative point of view, and aim to discover and enumerate different understandings that students have. While the question focuses on variables, I do include aspects of other concepts – such as type, value, object and assignment – within the scope of the research inasmuch as they define students' understandings of variables. The scope of this research is limited to object-oriented programming in Java.

1.2 Background

Poor results in introductory programming education have been widely reported, and students struggle to master even the most rudimentary programming skills (see e.g. [17]). Prior work suggests that many factors contribute to the problem, including the lack and slow development of problem-solving skills (see e.g. [27]) and poor understanding of basic programming concepts. This paper focuses on the latter. Du Boulay [6] notes that some students' difficulties are associated with their limited understanding of "the general properties of the machine [they] are learning to control, *the notional machine*". A notional machine for object-oriented programming is significantly more complex than one for procedural programming [22]. In object-oriented programming even a relatively simple concept, such as variable, may be more difficult to learn and understand as it has more complicated relationships with many other concepts and requires a more complex notional machine that can explain objects and references.

According to constructivist theory, beginning programmers construct different mental models of the underlying layers of abstraction [3]. Some student-constructed mental models will be viable, others will not. Many non-viable mental models arise out of students' incorrect assumptions or guesses about the notional machine that their programs are supposed to instruct. Programming teachers have the task of helping students construct viable models of the notional machine, and steering them clear of incorrect ones. It is useful to be aware of the educationally critical aspects of the notional machine and to know what pitfalls to look out for. This paper explores how introductory students understand a particular aspect of the notional machine: variables.

The paper is structured as follows. The next section introduces some related work on the phenomenographic research approach and on understandings of basic programming concepts. In Section 3, I describe the research setting and the methods I used. Section 4 presents the results and Section 5 discusses their implications. In Section 6, I look at future work possibilities, and Section 7 provides some concluding remarks.

2. RELATED WORK

2.1 Phenomenography and Variation Theory

Phenomenography [15] is an approach to research that investigates phenomena and people's relationships to those phenomena. Phenomenography posits that there are a number of qualitatively different ways in which a phenomenon is experienced by people and that these different ways are linked to each other. Within the community of CS education research, phenomenography has been applied, among other things, to studying beginners' understandings of programming in general (see e.g. [5, 7]) and students' understandings of specific computing concepts (see e.g. [8, 4, 24]).

To a phenomenographer, learning is characterized by the learner discerning new *dimensions of variation* [16]. Each critical aspect of a phenomenon, that is, an aspect that contributes to make the phenomenon what it is, is associated with a dimension of variation. Noticing different values along a dimension of variation leads to discerning the existence of the dimension, which in turn leads to a more sophisticated understanding of the phenomenon. Discerning relationships between values in a dimension, and between dimensions, leads to still deeper understanding. To take an example from computer science, Eckeral and Thuné [8] concluded that students' failure to reach sophisticated understandings of objects and classes is caused by failure to discern variation in critical aspects of the concepts. A student that sees objects only as pieces of code, and fails to see them as actors during program execution, would need to be shown different examples of relationships between a class description, object actions and resulting events in program execution.

Phenomenography does not prescribe a particular method for gathering or analyzing data. Nevertheless, traditions within the phenomenographic community contribute towards something that could be called a 'typical phenomenographic research methodology'. In a research project of this kind, interviews are used as a data collection method. Data collection is followed by or intertwines with qualitative data analysis. During analysis, the researcher, in dialogue with the data, delimits the phenomenon of interest. Different ways of understanding or experiencing the phenomenon are enumerated as an *outcome space* consisting of a (smallish) number of *categories of description*. Outcome spaces often take the form of a hierarchy or tree of categories related to each other.

The intention in phenomenographic research is not to point out which specific kinds of understanding each individual has, but to identify different ways in which a phenomenon can be understood, or experienced, on a collective level. Each category of description represents a partial way of understanding the phenomenon, and an individual person may understand the phenomenon in a number of the different ways represented by the categories of description. While the categories are logically connected to each other on a collective level, as defined by the researcher's interpretation of the data, an individual's understanding may not follow the same logic.

In the hard sciences such as computer science, there are concepts that are well defined. We can say that some understandings of these concepts are correct and others incorrect. According to Sorva [25], there are three approaches to dealing with correctness and incorrectness in phenomenographic studies that investigate students' understandings. The *equal approach* includes in the outcome space all undestandings that seem relevant, irrespective of correctness. Correctness plays no explicit part in delimiting phenomena and analyzing the data, and a discussion of the correctness is left for later, possibly to third parties reading the results. An example is Adawi and Linder's [1] work on understandings of heat in physics. Using the equal approach one can investigate both correct and incorrect understandings as long as one takes care in delimiting the phenomenon, and does not combine understandings of multiple separate (perhaps imaginary) phenomena into one outcome space. The *clean cut approach* focuses only on those aspects of understandings which the researcher deems correct as defined by the scientific paradigm or technical specification that provides the intended learning outcome. Incorrect aspects of understandings are excluded from the outcome space. Eckerdal and Thuné's [8] work, mentioned above, is an example of this approach. The clean cut approach makes delimiting the phenomenon less problematic, the downside being that it can not be used to investigate incorrect understandings. The *anchored approach* attempts a compromise between the other two approaches. In this approach, both correct understandings (as deemed by the researcher) and *partially incorrect understandings* are included in the outcome space. Partially incorrect understandings are understandings that extend correct understandings in incorrect ways. An example of the anchored approach is my own earlier work [24] on students' understandings of storing objects.

2.2 Understandings of Variables

Several prior studies have qualitatively explored understandings of variables and related concepts such as assignment.

Bayman and Mayer [2] studied beginners' interpretations of various statements in the BASIC language by asking introductory-level students to write plain English explanations of programs. They analyzed the resulting short descriptions of the semantics of BASIC statements, and list a number of related misconceptions. Examples: the statement `INPUT A` is taken to mean that the computer waits for the user to input a specific number or letter; the statement `LET D = 0` is interpreted to mean that the equation `D = 0` gets stored in the computer's memory.

Holland et al. [11] noted several misconceptions beginners have about objects. For example, students may conflate the concepts of object and class, or they may confuse an instance variable called `name` with object identity or with a variable referencing the object. Holland et al. [11] discuss the possible sources for these misconceptions and suggest potential pedagogical solutions. Their work is based on anecdotal but intuitively appealing evidence gathered while developing distance education courses.

Ragonis and Ben-Ari [21] report the results of a wide-scope, long-term, action-research study of high school students learning object-oriented programming. The study, which is based on a qualitative, constructivist approach, uncovers an impressive array of difficulties students have with object-oriented concepts (e.g. "constructors can only be used to initialize instance variables"). The authors categorize the misconceptions and other learning difficulties, and offer pedagogical advice. Their work centers around objects, methods and constructors, and coverage of variables is limited to objects' instance variables.

Sajaniemi and Navarro Prieto [23] investigated qualitatively different ways in which variables can be used in programs, according to expert programmers. They found that experts' knowledge about variable usage patterns can be described using *roles of variables*, which can be taught to novices to enhance learning. However, their work is concerned with algorithmic roles, a relatively high level of abstraction compared to the notional machine level that this paper is concerned with.

Ma et al. [13] studied introductory programming students' mental models of assignment. They gave a large number of volunteer CS1 students a test with open-ended and multiple-choice questions, and analyzed the answers qualitatively and quantitatively. A majority of students were found to have non-viable mental models of basic programming concepts; many did not even have a viable model of assigning primitive values to a variable. Ma et al. list a number of non-viable mental models of assignment, illustrating that there are problems both with understanding programming language syntax or semantics (e.g. assignment works from left to right) and with the underlying programming concepts (e.g. assignment stores an object inside a variable). Ma et al. [14], drawing on constructivism, suggest that cognitive conflict be used as a pedagogical tool against non-viable mental models.

3. RESEARCH SETTING AND METHODS

The results I present in this paper are part of a wider project that investigates students' understandings of program execution and related concepts. Earlier results from the same project, focusing on how students understand storing objects in memory, can be found in a previously published paper [24].

I chose phenomenography as research approach because phenomenographically obtained results can complement prior findings in two significant ways. First, phenomenography is well suited to the exploring of understandings on a collective level and discovering logical relationships between types of understanding. This can help us see the big picture and justify pedagogical solutions to learning problems. Second, phenomenographic analysis works well together with data collected from semi-structured interviews. Questionnaire and exam answers, which are analyzed in various studies, are often limited to searching for meaning in the specific words respondents use in an isolated sentence. Unexpected answers may be difficult to decipher, as noted by Ma [13] and others. If a student states in a questionnaire that `a = new Person("Jack")` mean "Jack is stored in position a", what does that really mean? In semi-structured interviews, it is possible to ask for clarifications and probe with follow-up questions, producing richer data about understandings on a conceptual level.

3.1 The Students

This study makes use of data from 17 interviews. Ten of these were done in Spring 2007 and have a relatively broad focus, covering many topics related to program code and execution. [24]. In Spring 2008, I added to the data pool seven further interviews that focus more closely on variables, as described below.

The additional interviewees were from a semester-long university course in introductory programming given in Spring 2008. The course teaches programming in Java in an objects-early way, yet without going deep into object-oriented design or complex object interactions. Apart from a few drawings at lectures, no tools visualizing the notional machine or computer memory are used in the course, and these topics are given rather little attention during the course. The course is taken by engineering students who are not computer science majors. The author of this paper (that is, the interviewer), while a teacher at the same department that gives the course, did not participate in running this course.

One third through the course, all students were required to complete an online questionnaire about their programming background and about their attitudes, experiences and workload during the course. Of the several hundreds of respondents, I selected a small subset and invited them for interviews based on this background questionnaire. In order to capture a wide range of qualitatively different understandings, I tried to maximize variation [20, p. 234] through an informal method: I hand-picked the invitees so that there were interviewees with different kinds of programming background (though most had no prior experience), different kinds of attitudes to programming and different experiences with the course.

I sent out fifteen invitations via email. I promised each student two movie tickets as a reward for an interview, and stressed that the interviews were not a part of the course and would not affect grading in any way. Seven of the invited students agreed to take part. The number of positive responses from invitees was surprisingly low but tentatively acceptable as I already had many interviews from the previous year and since there was a great deal of variation in the questionnaire answers of the seven new interviewees.

3.2 The Interviews

The interviews from 2008 are described below. The 2007 interviews were similar, differing mainly in that they covered a wider range of topics and did not focus specifically on variables. They are described in more detail in the earlier paper [24].

I interviewed the students roughly half-way through the introductory programming course they were taking. The interviews were done in Finnish; all interview quotes in this paper have been translated from the Finnish originals. Each interview began with a short generic discussion of what came to the student's mind when the word 'variable' was mentioned. After that, the bulk of the interviews revolved around the code of two simple Java classes that I showed to the student on paper. First, I showed them a class that only contained a main method that made use of a for loop and four integer variables to read in some values and print out computation results. The second class represented players of an imaginary game, with names and scores as instance variables. It contained a main method which created some player objects, manipulated their scores, and assigned player objects to variables in various ways. I asked the students to describe what they saw and how the programs worked. Follow-up questions concentrated on the variables and assignment encountered in the code (though those terms were not necessarily used by the student or myself). After each interview, I did some early analysis on what had been said and wrote it down. All interviews were recorded in audio.

Combining the data with the previous year's resulted in a pool of seventeen interviews. The early analysis suggested that an acceptable degree of saturation had been reached, so I did not recruit any more interviewees.

3.3 Data Analysis

I transcribed the relevant parts of each interview verbatim and added them to a pool of data. which I analyzed with the aim of discovering collective-level understandings. I started transcribing and analyzing data right after the first interview, so that the analysis could provide feedback and ideas for the rest of the interviews. In Section 2 above, three approaches to dealing with incorrectness of understandings in a phenomenographic study were identified. Of these, I used the anchored approach, meaning that in addition to correct understandings, I explored such incorrect understandings that extend correct understandings.

4. RESULTS

At the outset of the study, I expected to form a single outcome space describing qualitatively different understandings of the programming concept that the canonical term 'variable' refers to. However, it quickly emerged that where I had perhaps naïvely expected to find a single categorization of understandings, two separate categorizations would be needed. Numerous students experienced constructs such as `int number` and `Player p` as instances of two quite separate concepts. To produce more legible and usable results, I chose to delimit the phenomenon differently than initially planned and to break down the original research question into three subquestions.

1. In what different ways do CS1 students understand what a primitive variable is?
2. In what different ways do CS1 students understand what an object variable is?
3. In what different ways do CS1 students understand the relationship between primitive and object variables?

Three outcome spaces emerged from the analysis to answer these three subquestions. They are described in the subsections below.

4.1 Understandings of Primitive Variables

Table 1: Categories Describing Understandings of Primitive Variables

Category	Focus	Description
NAMEDVALUE	names in code	A variable is some kind of pairing of a name to a changeable value. It is characterized by a type, which restricts what the value can be.
PLACEFORVALUE	storing values	As NAMEDVALUE, plus: A variable is a typed place or slot located in the computer. It can be assigned a value, which it stores, and which can be accessed using the variable's name.
PLACEFORREF	references	As PLACEFORVALUE, plus: The value assigned to a variable is a reference to another location in the computer that stores some data value.
MATHVARIABLE	equivalence	As NAMEDVALUE, plus: A variable is a typed symbol which is equal to a value. It can be used in equations which declare relations between variables.

An outcome space with four categories describes understandings of primitive variables. Table 1 gives an overview of the categories, described in more detail and illustrated with quotes below.

Category: NAMEDVALUE

This category describes a general understanding that variables like `int number` are something that have to do with manipulating values in a program. Variable names correspond to values, which can change during program execution. Larry[1] is asked to identify variables from code, he lists identifiers of variables he sees:

> **Larry:** Well, there's `number` and `number2`...

Prompted to describe how he sees what a variable is, he explains:

> **Larry:** Well, you can give it different values and they change as the program is processed.

Larry notes that some variables are "variables in the traditional sense" as their values actually change, whereas others remain constant. However, he is at a loss to elaborate on what a variable is. His understanding is very vague and abstract.

> **Larry:** I don't know how to describe it any better. It's just... It's given some value.

The focus of awareness in this type of understanding is on names (or identifiers) in program code. Variables are seen as a way of naming and accessing values. They differ from each other in having different names and different values. Further variation is discerned in variables' values over time during a program execution. However, the relationship between the two key dimensions of variation – names and values – is very fuzzy. What it means that a variable "has" or "is given" a value is barely understood, but the practical consequence is that names can be used to manipulate values. No relationship between variables and memory locations is discerned.

Even on this very basic level of understanding, variation is discerned in variable types. Larry gives a fairly typical CS1 student's response when asked if he's seen variables in the course.

> **Larry:** Yeah, there are integer variables, and decimal variables, and then there's variables of the class `char` and of the type `String`.

Larry understands that type is a critical aspect of a variable (in Java), and later makes use of this knowledge when describing that variables are used to store values of a specific type in the given program. All the other, more complex understandings of primitive variables represented by the other categories of description extend this rudimentary type of understanding.

[1]Interviewee names changed.

Category: PLACEFORVALUE

This category of understanding extends the basic understanding of primitive variables described above. The focus in this category is on variables' use as storage. The relationship between a variable and its value, only vaguely hinted at in `NamedValue`, is understood as that of a storage and its contents. Paula explains:

> **Paula:** I understand a variable to be a compartment, whose contents change. Kinda like, you play around with the contents in different kinds of formulas, and then it gets the value that comes out.

Paula notes that the programmer can place data in variables, which help keep track of the data.

> **Paula:** I create a memory slot in which I place some thing, so that I can see it the whole time.

While the concept of computer memory may not be understood very well, it is understood that the computer is capable of storing data in various distinct locations. Location within the computer is discerned as a critical aspect of variables. Assignment to variables is understood as placing values in these locations, to be stored in variables. Chris, like Paula above, notes that there are places within the computer where values can be stored. He explains how a couple of assignment statements work:

> **Chris:** I have the vague notion that there are these locations in the memory of the computer... It puts [the values to the right of the assignment operator] in some location in memory.

Category: PLACEFORREF

This category of understanding extends PLACEFORVALUE and NAMEDVALUE. In this type of understanding, the concept of a reference is made focal and related to the concept of a (primitive) variable. As in PLACEFORVALUE, a variable is understood as storing data, but indirectly, through a reference. Each variable is characterized not only by a name, a type, and a 'storage slot' within the computer, but also by a stored reference which points to the actual value. It is perceived that each value is stored in memory in some location separate from the storage space corresponding to the variable itself. Noel captures this type of understanding in a nutshell as he explains the statement `number2 = number;`.

> **Noel:** In practice, that means they have the same value, but apparently it goes so that `number2` isn't given the value of `number`, but rather `number2` refers to the integer value. I'm not sure, but I think that when `number` changes, then `number2` will change at the same time.

Category: MATHVARIABLE

This category of understanding extends NAMEDVALUE. As in that category, variables are seen as having types, names and values. No additional dimensions of variation are discerned. Instead, this understanding involves a particular interpretation of the relationships between two critical aspects of variables: names and values. For Otto, a variable *is* the value:

> **Otto:** A variable is a single number or a sequence of characters.

A variable's name is understood to be a symbol for an unspecified value, much like a variable in mathematics.

> **Otto:** [A variable is] any letter, which we decide is something else.

Occurrences of the variable's name are considered logically equivalent to the variable's value. Variables receive their values logically through equations established by statements in a program. Consider this code.

```
number2 = number;
number = keyboard.nextInt();
result = number2 + number;
```

At the start of the interview, Quentin noted that a variable in programming "is like a variable in an equation". Let us see what he makes of the code above.

> **Quentin:** You first input `number`. Then it calculates `number` plus `number`. I mean, you give a number and it basically multiplies it by two.

Quentin is unable to explain why he came to this conclusion. Otto, however, concludes the same from reading the code, and explains his reasoning as follows.

> **Otto:** I think they [`number` and `number2`] are always equal since it says `number2 = number`.

That this type of understanding of variables as mathematical symbols may be related to a number of non-viable understandings of related Java programming concepts such as assignment and the program execution sequence. The order of the lines is irrelevant to Quentin and Otto, since they read them as a mathematical system of equations rather than an execution sequence, and expect the computer to do likewise. Quentin is explicit about this later, after I point out his earlier error to him.[2]

> **Quentin:** I didn't pay attention to the order. I just read it as if the program had the brains to see [`number2 = number`] above. That it would have the brains to go read the bit from above.
> **Interviewer:** Some others have seemed to think that it goes like in math, so that "x equals y" is like a declaration that x is the same thing as y...
> **Quentin:** Exactly! That's how I thought!

[2]I tried to fix some incorrect understandings afterwards.

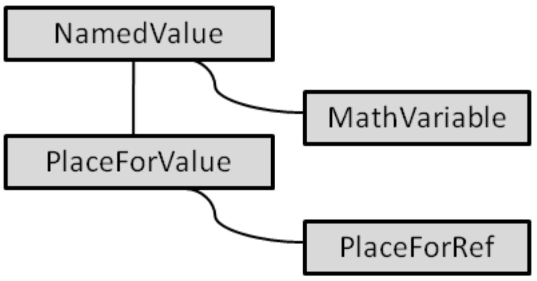


Figure 1: Relationships between categories of understandings of primitive variables. Each line indicates that the category below extends the category above.

Relationships between Categories

Relationships between the categories listed above are illustrated in Figure 1. NAMEDVALUE is a simple, partial understanding of primitive variables. It is extended by PLACEFORVALUE, which is richer in that it separates the concepts of variable and stored value. PLACEFORREF and MATHVARIABLE are partially incorrect understandings, or overextensions, in the sense that they are characterized by a focus on aspects (namely, equations and references) that are not part of the orthodox definition of variable. The partial incorrectness is depicted in the diagram by the way these two categories 'branch out' to the right.

4.2 Understandings of Object Variables

Another outcome space with four categories of description emerged to describe understandings of object variables. Table 2 gives an overview of the categories, which are described in more detail and illustrated with quotes below. The reader may note that some of the labels given to categories of description in this outcome space are similar or identical to the labels in the first one. This is neither coincidental nor accidental. It is meant to underline the similarities between the two outcome spaces.

Category: NAMEFORTHING

This category describes a rudimentary understanding of object variables. A variable name is, as Larry puts it, a name within the program code for an object. Apart from name and type, very little is understood about what a variable is. Variation is discerned in that variables are associated with different "things" or objects, but the relationship between the two is unclear apart from the fact that a name is used to work with objects. Variable declarations and identifiers are simply necessary in practice to get programs working. Larry explains what the declaration `Player fourth` means:

> **Larry:** Here we create a new object. [Stops to think.] Or I dunno, at least we provide the opportunity to do it, but don't really create it. [...] You have to put this there first in order to be able to... If this didn't exist then at least you wouldn't achieve what you were supposed to achieve. So you have to have this.

All the other categories in this outcome space extend this very simple category of understanding.

Category: PLACEFORVALUES

This category extends the basic understanding of object variables described above. The focus is on variables' use as storage. The

Table 2: Categories Describing Understandings of Object Variables

Category	Focus	Description
NAMEFORTHING	names	A variable is a name which can be used to manipulate an object or 'thing' in a program.
PLACEFORVALUES	storing values	As in NAMEFORTHING, plus: A variable is a typed place or slot located in the computer. A set of object properties can be assigned to be stored in it.
PLACEFORREF	references	As in PLACEFORVALUES, plus: The value assigned to a variable is a reference to another location in the computer that stores an object.
PROPERTY	object	As in NAMEFORTHING, plus: A variable is not a separate entity, but an aspect of an object. Its name is one of the object's properties, akin to the object's instance variables.

relationship between a variable and an object, barely discerned in `NameForThing`, is understood as that of a storage and its contents.

Mike notes that the command `Player third` defines a variable and and that `third = new Player("Cecilia")` creates a new player object with the given name and a score of zero. Here he explains the notion of variables that store objects:

> **Mike:** These [variables of type Player] can receive as their values both a variable of type String and the score. I mean, two values of different types. Whereas [these primitive ones over here] receive only numerical values.
> **Interviewer:** Right. They differ in that they have different types?
> **Mike:** Yeah, and in that the players contain *multiple* kinds of information.

The computer is understood to contain places for storing objects' properties (e.g. player objects' names and scores) as composite chunks of data. Variables correspond to these storage slots, and assigning values to variables means storing object data in a new place. Otto explains the statements `fourth = second; third = second;`

> **Otto:** The second player's data is copied into the fourth player. Likewise into the third. So the fourth, third and second player will be equal.

Category: PLACEFORREF

This category is an extension of PLACEFORVALUES. It builds upon the idea of storing data in variables, incorporating the focal concept of object reference. A variable is understood as storing data indirectly through a reference. Each variable is characterized not only by a name, a type, and a 'storage slot' within the computer, but also by a stored reference which points to the data of an object. It is perceived that each object is stored in memory in some location separate from the storage space corresponding to the variable itself. Brad (who later demonstrates a practical understanding of what happens in reference assignment) comments on a statement which creates an elevator and assigns it to a variable named `testElevator`.

> **Brad:** I think this `testElevator` itself only stores, like, the memory location. . . which refers to the object, to where it is in memory.

In this type of understanding, assignment to a variable is also perceived in terms of references. It is understood that assignment only copies references and that a single object can be referenced by multiple variables.

Category: PROPERTY

This type of understanding is an extension of NAMEFORTHING. No additional dimensions of variation are discerned. Rather, this category differs from NAMEFORTHING in how the relationship between variable declarations and objects is understood. Variables are not understood as a separate construct in their own right at all. Rather, they are seen as an aspect of a larger "object concept". Manipulating objects requires a name of some kind, and all objects have one. This explains the need for certain definitions in the code (e.g. `Player p`), which, as in NAMEFORTHING above, are seen as necessary in order to make use of objects. Below, Ian explains his idea of what constitutes the data for an elevator object. (The class `Elevator` has two instance variables, `floor` and `topFloor`).

> **Ian:** [The data for one object is] the `this.floor` of the elevator and the top floor... and then the name, I guess... Yeah.

Throughout the interview, Ian is consistent in that he treats variable names as an object property. Greg has a similar view. He explains object assignment as follows (an elevator object is assigned to a variable named `test1`).

> **Greg:** Since it had been assigned the name `newElevator`, it changes it to `test1`. [. . .] And I suppose its floor also changes[.]

For Greg, assignment means changing an object's attributes, including both the instance variables and the "object's name", i.e., a property that defines what the object has been assigned to.

Relationships between Categories

Relationships between the categories in the second outcome space are illustrated in Figure 2. NAMEFORTHING is a basic, partial understanding of object variables. It is extended by PLACEFORVALUES, which is richer in that there is a clearer picture of a variable as data storage. In this partial understanding, the important role of references is not understood; however, references are focal in the further extension PLACEFORREF. NAMEFORTHING is also extended by the partially incorrect category PROPERTY, in which objects are meshed into the concept of variable.

4.3 Understandings of the Relationship between Primitive and Object Variables

A third outcome space describes qualitatively different understandings of primitive and object variables as a whole. The categories represent different ways of seeing the connection between primitive and object variables. The outcome space describes the relationship between the two first outcome spaces from the collective point of view of the students. Table 3 gives an overview of the categories, which are described in more detail below.

Table 3: Understandings of the Relationship between Primitive and Object Variables

Category	Focus	Description
DIFFERENT	code	There is a mechanism for storing and assigning primitive values, and another for objects. These mechanisms have superficial similarities, but are fundamentally different from each other.
SAME	variable concept	As in DIFFERENT, except that: There is a connection between the two kinds of variables. They are both examples of a single mechanism for storing data, which works exactly the same way for both primitive values and objects.
SAMEBUTDIFFERENT	memory	As in SAME, except that: The relationship between variable value and data is different for primitive as opposed to object variables.

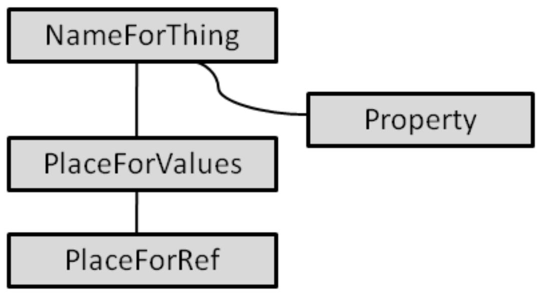


Figure 2: Relationships between categories of understandings of object variables. Each line indicates that the category below extends the category above.

Category: DIFFERENT

In this category, a dimension of variation is discerned concerning the data manipulated by Java programs. Some of it is primitive, some of it is objects. However, a relationship between the ways in which these two kinds of data are used is discerned only barely or not at all. What awareness there may be of a connection between primitive and object variables focuses on superficial aspects of program code such as the use of the assignment operator in both cases. The mechanisms used for storing and assigning primitive values, on one hand, and objects on the other, are seen as two fundamentally different things.

This category is characterized by different understandings of primitive variables than of object variables. For instance, Keith thinks of primitive variables as storage space for values (PLACEFORVALUES, Subsection 4.2) but his understanding of object variables is fuzzier (much like in NAMEFORTHING, Subsection 4.2). When I prompted him directly for any connection between the two kinds of similar-looking constructs in the code, Keith still could not think of anything apart from the fact that an object can contain variables. I then wrote down this improvised partial class definition for him to comment on:

```
class Match {
  private Player champion;
  private Player challenger;
```

Despite pointing out that he has seen something similar in the course, Keith was puzzled by the code and unable to figure out what it could possibly mean.

> **Keith:** Why does it say Player in there? In the other class [Player], we had *variables*. Like private int or private String[3].

[3]Keith, like many Java programming novices, groups Java strings with primitive types rather than with objects.

While there are surely many factors contributing to Keith's understanding, it is consistent with the sharp distinction that he makes between primitive variables – which are the only thing he calls 'variables' – and object variables. For Keith, it is natural that there are places in the code where you need 'variables', and he cannot begin to understand the use of objects instead.

Category: SAME

This category extends DIFFERENT with a clearer relationship between primitive and object variables. The focus here is on a single, unifying concept of variable, which both are examples of. Quentin comments on what `Player first` and `Player second` are.

> **Quentin:** A variable, that's what it is. Those are variables, too, in principle.

Syntactical similarity in how primitive and object variables are used in code is not perceived as coincidental. Rather, the same operations are applicable to both kinds of variables. They differ from each other only in terms of the type of data that they store, as illustrated by the quotes from Mike's in Subsection 4.2.

This type of understanding is characterized by a similarity of understandings of primitive variables compared to object variables. The same rules are understood to apply to both. Mike, for instance, thinks that all kinds of variables are storage slots for the actual primitive or object data. Conversely, Noel, quoted in Subsection 4.1, thinks that all Java variables are meant for storing references to the actual data.

Category: SAMEBUTDIFFERENT

This category further extends SAME. Again, it is understood that there is a more generic concept of variable. Some variation is discerned in how object and primitive variables work, however. Paula does a good job at explaining how she sees the relationship between these two kinds of variables.

> **Paula:** In a way they are the same thing, since we're talking about a quantity of memory inside the computer's memory, from the physical point of view. But – if I've got it right – when we have these [primitive] variables here, what we store in that memory slot, or that span of memory, is that very data. But when we have these objects, then the memory slot only stores an address of the object, which is somewhere else. But physically, it's the same thing. . . a location in the memory of the computer.

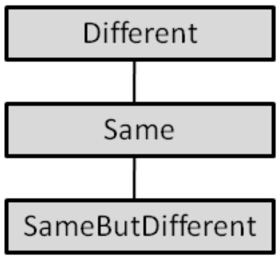


Figure 3: Relationships between understandings of the relationship between different kinds of variables.

Relationships between Categories

Relationships between the three categories are illustrated in Figure 3. The category DIFFERENT is the least sophisticated understanding in which barely any relationship between primitive and object variables is discerned. It is extended by the richer category SAME, where a strong connection between the two is made. That understanding is further enhanced in SAMEBUTDIFFERENT, where variation between the two kinds of variables is discerned despite the fact that they are both seen as examples of the same concept. This last category represents the richest type of understanding of variables that I found in this study.

5. DISCUSSION

5.1 Variables Misunderstood

The outcome spaces presented in the previous section provide insight into the understandings that fledgling Java programmers have of variables. Particular non-viable understandings from these outcome spaces have surfaced in literature before (see [6, 13, 2]). However, this paper brings together – and relates with each other through phenomenography and variation theory – understandings of both object and primitive variables in order to provide an overall picture of understandings of variables in an object-oriented context.

My results confirm once again the constructivistic observation that people create various kinds of mental models of programming concepts and of how the computer makes programs work. Even the concept of variable, which is fundamental to most types of programming and crucial for building Java programs, is understood partially or incorrectly in many different ways. Of the various different understandings discovered in this study, only a few are viable and complete enough for writing object-oriented programs of any significant complexity. It is not surprising that CS1 students have difficulties with the complexities of object-oriented programming if their understanding of basic constructs is shaky or incorrect. The division of data types in primitive and object types, as well as the interplay between the concepts of variable, object and reference, contribute to make Java variables a challenging concept to master.

The third outcome space, from Subsection 4.3 above, brings to the fore the fact that students do not always delimit concepts in the ways teachers might like them to, and demonstrates that the relationship between primitive and object variables is a problematic issue in teaching Java programming. The outcome space highlights some key challenges in learning about variables. As illustrated by the category DIFFERENT, these two kinds of variables are seen by some learners as being two completely distinct constructs, despite superficial similarities in syntax. Failing to discern a generic concept of variable makes it hard to comprehend programming literature and teaching. In terms of variation theory, the category DIFFERENT can be explained by a failure to perceive that two kinds of variables are values along the same dimension of variation. Such an unsophisticated understanding may derive from a poor understanding of the concept of data type. It is also partially explained by students' tendency to construct excessively narrow 'rules' for programming, which constrain the ways in which a programming construct can be used [9]. An example of a rule could be "Variables are meant for storing numbers and such (whereas composite objects are dealt with quite differently)." Learners with this type of understanding are likely to be at greater risk of constructing additional narrow rules. For instance, the misconception "Objects can't be the values of attributes" (reported by Ragonis & Ben-Ari [21] and echoed by Keith in Subsection 4.3 above) is more appealing intuitively if primitive and object variables are seen as two completely distinct constructs.

The category SAME alerts us to another type of overgeneralization: all variables work exactly the same way and have the same kind of relationship to the data they are associated with. Such an understanding is in practice often accompanied by either the incorrect assumption that primitive variables store references (see PLACEFORREF, Subsection 4.1) or that object variables store actual object data (see PLACEFORVALUES, Subsection 4.2). Either way, such an incomplete understanding is non-viable as soon as the student needs to write programs that manipulate, say, both primitive and object parameters.

Some issues are linked to primitive or object variables specifically. The category MATHVARIABLE is explained by students' exposure to mathemathical variables and equations, by Java's math-like syntax and by constructivist theory. It matches anecdotal evidence on learning difficulties related to the assignment statement and is reminiscent of the misconception reported by Bayman and Mayer [2], where the the statement `LET D = 0` is interpreted as storing an equation in memory. A student with such an understanding of primitive variables will struggle until they come to understand the storage aspect of variables in programming. On the objects side, the category PROPERTY shows a conceptual conflation of objects and the variables referencing them. This category, which meshes together two concepts, the outcome space of understandings of variables 'touches' the outcome space from my previous project [24], where a similar category emerged to describe understandings of objects in memory. Clearly, understandings like PROPERTY and the vague NAMEFORTHING can lead to a variety of practical problems for learners of programming. If no clear distinction is drawn between variable and object, the concept of reference remains fuzzy at best, and object assignment becomes difficult or impossible to grasp.

5.2 Pedagogical Implications

Learning about Java variables is problematic. Teachers must take care to structure courses so that enough time is reserved to make sure that basic concepts are properly understood. This is particularly important on introductory programming courses that teach object-oriented programming and its more complex notional machine. Otherwise, students are cognitively overloaded as they have to learn more advanced topics while still struggling to master the basics. Relevant parts of the notional machine should be taught along with each code construct to decrease the chance of students constructing non-viable models of the runtime system. (E.g., references should be discussed as soon as object variables are.)

The three outcome spaces highlight educationally critical aspects of variables, which teachers should emphasize to students on CS1 courses. Further, teachers can be helped by an awareness of the kinds of partial and incorrect understandings that students have, as prior understandings affect the ways in which students process additional information. Sorva, drawing on variation theory, writes: "As programming instructors, we need to draw students' attention to the important aspects and variation in [the correct critical] aspects, and underline their importance where possible. [. . .] On the other hand, students may also mistakenly focus on irrelevant variation and mistake it for critical variation. Here, our task is to draw the students' awareness away from the irrelevant focus." [24]

The first outcome space (Subsection 4.1) provides hints to an teacher facilitating the development of students' understandings of variables. Instruction should be designed to aid the development of an understanding of variables as storage over the simplistic understanding described by the category NAMEDVALUE and the incorrect understanding represented by MATHVARIABLE. An emphasis is needed on the variation in variables' memory locations, and the separateness of variable and value. Instruction should be explicit about the similarities and differences between math variables and programming variables, and equations and assignment statements, respectively. Assignments requiring students to reflect on the program's execution sequence may help.[4] Engaging visualizations of a part of the notional machine – variables and values in memory – could be helpful and should be designed so that they can draw attention to the critical aspects. Visualizations and metaphors could likewise play a role in avoiding or dispelling the incorrect understanding PLACEFORREF: students can be shown that assignment makes multiple copies of the same value in the computer's memory, and that references play no part in this. It is also easy to come up with examples that demonstrate how changing primitive variables' values does not affect other variables.

Object variables stretch learners further. A vague understanding of the NAMEFORTHING variety is barely if at all sufficient for object-oriented programming. Steps must be taken to prevent learners from coming up with incorrect interpretations like PROPERTY. Various pedagogical methods, examples and tools can be used. Irrespective of which method is used, it is important to make sure that students are able to draw a line between the concepts variable and object. These concepts should be introduced one by one. At least initially, declaring a variable and assigning an initial value to it should be done separately, not with a single statement. Once again, engaging visualizations and metaphors could help, provided they are designed to highlight the educationally critical variation. Examples should be carefully chosen to illustrate the critical aspects of variables and objects, and the ways in which the two are distinct: variables have memory locations, objects have their own locations, variables' have names that are not properties of any object, variables can be used to access – but are not the same as – objects.

Introducing variation along a particular dimension while keeping another aspect constant can be a very efficient educational tool [16]; students should be shown examples that demonstrate how multiple variable names can reference the same object and how a single variable name can reference multiple objects in succession. To develop an understanding richer than PLACEFORVALUES, students also need the concept of reference. Example programs that demonstrate the non-viability of PLACEFORVALUES are not hard to come up with; the main challenge is to engage the student in thinking about what underlying mechanisms (i.e., what kind of notional machine) make a particular piece of code work as it does.

[4]In one case, having noted that one student interpreted assignments as equations and variables as math symbols, I then asked the student to think about how the program's behavior would change if the lines were reordered. This simple question gave pause to the student, who then had a 'eureka moment' and later explained how he felt this had helped him greatly with the whole course.

The third outcome space presents teachers with a twofold challenge. On the one hand, there is a need to emphasize the relationship between primitive and object variables. Students' attention should be drawn to the facts that both kinds of constructs are called variables, that they both represent fixed-size memory locations meant for storing values, and that the same Java syntax is used to assign to and read from both. The concept of data type and the subconcepts of primitive and object type must be stressed. We need to clarify to students how all variables fundamentally store the same thing, i.e., bit sequences that are interpreted as values that are in turn interpreted in some way by the runtime system. If visualization tools are used, they should visualize primitive and object variables in the same way. On the other hand, the variation in how the two kinds of variables relate to data needs to be stressed. Students need to be shown that the only difference between primitive and object-typed variables is a result of their data types: how the notional machine makes use of primitive values differs from how it uses reference values. Only through a viable model of these mechanisms can students understand how primitive variables and object variables are "the same but different".

The CS1 course whose students were interviewed in this study teaches little about either the physical computer nor a more abstract notional machine. Instruction focuses largely on code-level abstractions. This is likely to be at the root of some of the student difficulties discovered, and might be helped by making the computer more explicit in exercises and classroom instruction. Ma et al., reporting students' poor performance in describing assignment statements, asked:

> "Is the knowledge on the memory mechanism of a computer necessary for early programming learning, and should programming instructors teach the knowledge explicitly at an early stage of programming education?" [13]

Not all students fail to cope. Learners come up with mental models even in the absence of a taught conceptual model, and some such models are adequate for the purposes of a CS1 course. Let us listen to Jenny explain how she thinks about memory.

> **Interviewer:** So, there are 'places' in memory?
> **Jenny:** Yeah, like slots, like street addresses, and something lives at each address. [. . .] Like, let's imagine that the object lives in a house, and then in that house there are rooms, which contain some other stuff like these [object attributes]. [. . .] A reference is like the address of a house.

Jenny also understands how primitive and object variables work (the only difference being that the latter stores an 'address'). Prompted for the source of this extensive metaphor, Jenny explains that someone had told her that the computer's memory is "a set of addresses" and she "sort of took it further from there". She has done well (and been a bit lucky?) to come up with a viable way of thinking about memory, despite receiving little instruction on the topic. However, as evidenced by examples above, many other students struggle. The development of an understanding of code is undermined by nonviable models of the underlying machine. In light of results such as those reported here and elsewhere (see [13]), teaching about the memory mechanism in CS1 courses is certainly worth a try, especially if object-oriented programming is also taught. I will return to this topic under Future Work, below.

Finally, this study highlights the importance of terminology in teaching programming concepts. Many students experience great conceptual confusion when learning programming, and their definitions of terms that they use are often unorthodox (as I noted once again firsthand when trying to find out what the students thought a variable is rather than what they meant by the word 'variable'). Not only do students use terms that are patently incorrect, but they also delimit concepts in different ways than the teacher. For a given student, the word 'variable' might mean a primitive variable, a local variable, a math-style symbol, or something yet different. Unless the teacher knows how his students understand the basic programming terms, it is likely that much of what the teacher says or writes will be understood quite differently than he hoped. Teachers must take great care to try and find out how their students understand terms and concepts. This is a tricky task, as CS1 students often are not very good at defining terms. Teachers themselves must be careful to use terms consistently. (Referring to object variables as 'objects' is common but misleading!) A shared terminology should be established early on in a course and students' interpretations of terms should be examined early.

6. FUTURE WORK

While the nitty-gritty of a computer or a virtual machine are (arguably) not an appropriate topic for a CS1 course, the capabilities of an abstract notional machine or runtime system (arguably) are. Gries [10] advocates the use of a consistent higly abstracted visual metaphor for variables, objects and references. He warns that "introducing computing concepts in terms of the computer can create unnecessary and confusing detail". It is easy to agree that using high-level abstractions in teaching programming is very useful. Nevertheless, the use of easy-to-understand representations of the computer's role as executor of programs is likewise worth exploration, as understanding program execution on a slightly lower level of abstraction could surely solidify learners' understandings. The challenge is to find a suitable level of abstraction that clarifies the semantics of program code without getting bogged down in technical minutiae, and a pedagogically sound way to teach on this level of abstraction.

Program visualization [26] has been claimed to aid learning by making underlying abstractions explicit. Various program visualization tools exist. For instance, Jeliot 3 [18] visualizes many aspects of Java program execution including variables and memory allocation.

Where Jeliot 3 perhaps comes short is engaging students. A visualization tool should engage learners in interactions [19]. Actively applying themselves to interact with a visualization could help students discern new variation in concepts such as variable. It can also induce cognitive conflict between a student's existing non-viable models and the conceptual model in the visualization. Ma et al. [14] experimented with a tool that requires learners to predict the values of variables before showing actual variable values in a way that promotes cognitive conflict. The TRAKLA2 system features interactive algorithm tracing exercises, which engage students by requiring them to indicate in detail how a given algorithm affects a given data structure visualization [12]. Drawing on all these ideas, we can envision a new tool that would not only visualize the notional machine, but engage students to predict how the notional machine works on a given piece of code. Students would have to predict and indicate how variables are created and given values, where references point, and so forth. Drawing on data such as that reported in this study, the tool and example programs could be designed so that they aid the discernment of critical variation and dispel specific incorrect understandings. Apart from variables, a number of other notional machine features could also be visualized. Students could be required to use the tool to predict allocation and deallocation of stack frames, expression evaluation sequences, etc. Creating such a tool presents an intriguing and promising future work challenge.

7. CONCLUSIONS

In this paper, I have described introductory students' understandings of Java variables as hierarchical sets of categories. My results suggest that even this basic programming concept is challenging to many students, especially in the context of object-oriented programming that makes use of object references. These findings complement and relate to each other earlier findings of problems with the concept of references and assignment. In particular, my results highlight several difficulties students have in discerning both the differences and the similarities between concepts (programming variable vs. math variable, object vs. object-valued variable; primitive vs. object-valued variable). It is crucial that teachers take into account that students have very different understandings of concepts. Drawing on the outcome spaces, teaching materials and tools can be developed to explicitly address potential pitfalls and highlight educationally critical variation to students. A software tool, which would engage students to interact with and manipulate a visualization of a notional machine, suggests itself as an intriguing avenue for future work.

Acknowledgements

Thanks to Lauri Malmi and Anders Berglund for their many comments on this work.

8. REFERENCES

[1] T. Adawi and C. Linder. What's hot and what's not: A phenomenographic study of lay adults' conceptions of heat and temperature. In *The 11th EARLI conference*, 2005.

[2] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, 1983.

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

[4] A. Berglund. *Learning Computer Systems in a Distributed Project Course. The what, why, how and where*. Uppsala dissertations from the faculty of science and technology 62, Uppsala University, Sweden, 2005.

[5] S. Booth. *Learning to program: A phenomenographic perspective*. Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden, 1992.

[6] B. Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[7] A. Eckerdal and A. Berglund. What does it take to learn 'programming thinking'? In *Proceedings of The First International Computing Education Research Workshop*, pages 135–143, 2005.

[8] A. Eckerdal and M. Thuné. Novice Java programmers' conceptions of "object" and "class", and variation theory. *SIGCSE Bulletin*, 37(3):89–93, 2005.

[9] A. E. Fleury. Programming in Java: student-constructed rules. *SIGCSE Bulletin*, 32(1):197–201, 2000.

[10] D. Gries. A principled approach to teaching OO first. *SIGCSE Bulletin*, 40(1):31–35, 2008.

[11] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. *SIGCSE Bulletin*, 29(1):131–134, 1997.

[12] A. Korhonen, L. Malmi, and P. Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm

simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

[13] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating novice programmers' mental models. `http://www.cis.strath.ac.uk/~linxiao/TechReport2006.doc`, 2006.

[14] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood. Using cognitive conflict and visualisation to improve mental models held by novice programmers. *SIGCSE Bulletin*, 40(1):342–346, 2008.

[15] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Associates, 1997.

[16] F. Marton and A. Tsui. *Classroom Discourse and the Space of Learning*. Lawrence Erlbaum Associates, 2004.

[17] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant", C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin*, 33(4):125–180, 2001.

[18] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 373 – 376, Gallipoli (Lecce), Italy, May 2004.

[19] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003.

[20] M. Q. Patton. *Qualitative Research and Evaluation Methods*. Sage Publications, 3rd edition, 2002.

[21] N. Ragonis and M. Ben-Ari. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, 15(3):203 – 221, 2005.

[22] J. Sajaniemi and M. Kuittinen. From procedures to objects: What have we (not) done? In J. Sajaniemi, M. Tukiainen, R. Bednarik, and S. Nevalainen, editors, *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group*, pages 86–100, University of Joensuu, Department of Computer Science and Statistics, 2007.

[23] J. Sajaniemi and R. Navarro Prieto. Roles of variables in experts' programming knowledge. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 145–159, 2005.

[24] J. Sorva. Students' understandings of storing objects. In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 127–135, Koli National Park, Finland, 2007. ACS.

[25] J. Sorva. Investigating incorrect understandings of a CS concept. In *Second Nordic Workshop on Phenomenography in Computing Education Research*. Uppsala University, 2008.

[26] J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[27] L. E. Winslow. Programming pedagogy – a psychological overview. *SIGCSE Bulletin*, 28(3):17–22, 1996.

Diagnosing Learners' Problem Solving Strategies Using Learning Environments with Algorithmic Problems in Secondary Education

Ulrich Kiesmüller
Didactics of Informatics
University of Erlangen-Nuremberg
Martensstr. 3
91058 Erlangen, Germany
+49 9131 8527936
Ulrich.Kiesmueller@cs.fau.de

ABSTRACT

At schools special learning and programming environments are often used in the field of algorithm. Particularly with regard to informatics lessons in secondary education they should help novices to learn the basics of programming. In several parts of Germany (e. g. Bavaria) these fundamentals are even taught in the 7th grade, when pupils are 12 to 13 years old. Age-based designed learning and programming environments such as Karel, the robot and Kara, the programmable ladybug, are employed there, however learners still underachieve. One possible approach to improve both teaching and learning process is specifying the knowledge concerning the learners' individual problem solving strategies, when they create their solutions in consideration of the solution attempt's quality.

A goal of the research project described here is being able to identify and categorise several problem solving strategies automatically. Due to this knowledge learning and programming environments can be improved which will optimise the informatics lessons, in which they are applied. Therefore the environments must be enhanced with special analytic and diagnostic modules, whose results can be given to the learner in the form of individualized system feedback messages in the future.

In this text preliminary considerations are demonstrated. The research methodology as well as the design and the implementation of the research instruments are explained. We describe first studies, whose results are presented and discussed.

Categories and Subject Descriptors

K.3.2 [**Computer And Education**]: Computer and Information Science Education – *computer science education, curriculum, self-assessment*.

General Terms

Algorithms, Measurement, Performance, Human Factors, Languages.

Keywords

Secondary Computer Science Education, Didactics of Informatics, Problem Solving Process, Algorithms, Kara, Tool-Based Analysis.

1. MOTIVATION

Teaching programming and basic ideas of algorithm often causes problems for the learners. This is indicated by the high number of college drop outs in CS1 courses and bad marks in Informatics at school. In order to minimize these problems programming is taught without writing code. Visual programming environments such as Alice [3] and Scratch [9] are employed in secondary and higher education in many countries.

Teaching the basic principles of algorithms is fundamental in the field of secondary Informatics education [14]. In this case learners are also given special didactically reduced, text-based or visual programming languages to make their first steps in programming. In some federal states of Germany (e. g. Bavaria), the basics of algorithms are already taught in the 7th grade (age 12 to 13 years). Here age-based learning and programming environments, such as Karel, the robot [11] and Kara, the programmable ladybug [12] are used.

Learners are motivated by the design of these learning environments and enabled to solve even complex tasks after only a few lessons because of the simple learnability and usability. Testing their solution attempts by program running often produces system error messages of the environments. The pure technical messages cause new questions during the problem solving process, consequently these learners need their teacher's help. If he/she only gives technical support and corrects the last step to reach the correct solution as fast as possible there is no difference to the system error messages.

What is also not an aim of Didactics of Informatics is to force the learner to copy exactly the sample solution at this point. All these variations lead to dependence, to further need of help and finally to aimless changing of the problem solving strategy and frustration. To avoid this, the help for the learner should be focused on his individual way of proceeding.

The pupils should be met where they are, which means the teacher will not only examine the actual solution attempt, but will ask the learners which method they applied to find a solution. Thus he/she does not only give technical support and correct the last step but tips the learner off how to reach the solution as cleverly as possible. Therefore it is necessary to get more knowledge about the learners' individual way of proceeding when constructing a solution. If the learning environment identifies the problem solving strategy automatically, it will be possible to give individualized feedback to the learner (with special consideration of the respective solution quality).

This encourages independence from teachers during learning and also increases the learners' motivation. In addition to that it enlarges their learning competence as a positive side effect. Thus it contributes to "life long learning" because the pupils have to deal with arising problems on their own and stick to an individual problem solving strategy for the whole problem solution. This will improve learning and programming environments and enhance the quality of the learning processes of informatics.

To reach this goal special research and diagnostic modules, which identify the problem solving strategy and create individualized system messages, have to be added to the learning and programming environments applied.

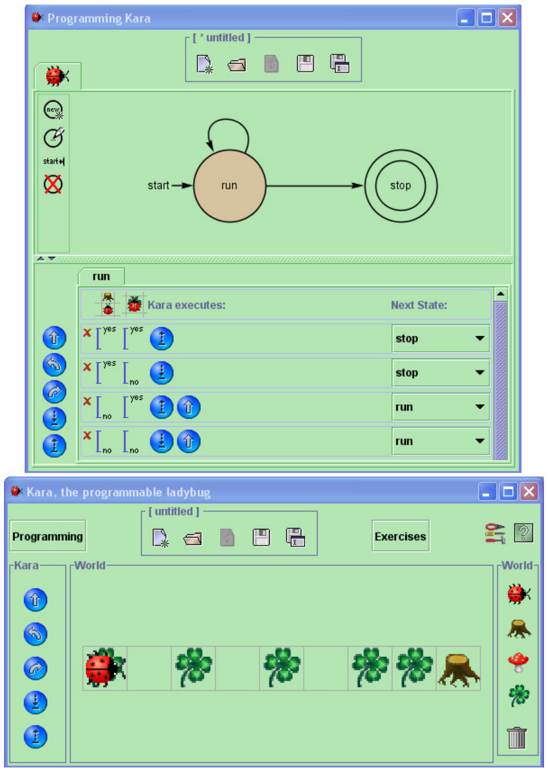


Figure 1. Screenshots of the Kara environment top: programming window – bottom: Kara world

2. THE KARA ENVIRONMENT

Kara [5] is an educational software system, which enables a learner to control a virtual ladybug based on finite state machines. Especially developed for programming novices it makes them learn the basic control structures such as command, sequence, conditional branch and iteration. At this the automata terminology need not appear to solve the tasks that have been set. Kara is placed in a chessboard-like world (see Figure 1) with fixed obstacles (i.e. trees) and movable objects (i.e. cloverleaves). It can turn left or right, move a single step ahead, lay down leaves and collect them again or push mushrooms (subsequently called *commands*). All commands can be combined in *sequences*. Furthermore Kara possesses sensors to test, e. g. if there is a cloverleaf on or a tree in front of the current position. With the help of these sensors *branching* of the program flow can be achieved. With the information, which state will be the next in the program flow (given by the transitions), *iterations* are realized.

One of Kara's typical tasks is navigating through a "forest" of trees and collecting leaves. In contrast to their possibilities in Karel, the robot, learners can program the system in a pure graphical manner. To cause the ladybug to fulfill a certain task, a learner has to identify the states needed and to specify the transitions with the help of sensors and commands. A survey among students has shown that "[…] Kara had allowed them to focus on problem solving, on the logic and the correctness of their programs, without being distracted by the environment or by the textual syntax of a 'real-world' programming language" [5].

3. PRELIMINARY CONSIDERATIONS

3.1 Learner-System-Interactions

In preliminary studies several subjects solving typical tasks of the Kara stuff were observed by a human researcher. The following interactions result from the recorded learner-system-interactions and some theoretical preliminary considerations. They are relevant to the problem solving process and therefore have to be recorded in subsequent studies (in the diagrams of section 6.1 used phrases are emphasized):

- editing and changing the final *state* machine – equivalent to the problem solving structuring
- creating and editing the conditions and *branch*es in consideration of the results of Kara's *sensor*s – equivalent to a fine structuring in several sub-problems
- editing *transition*s – each created branch automatically causes a transition, so the first appearance of a certain transition ranks among the fine structuring in several sub-problems – in all other cases: loops, branches, sequences of commands' sequences are modeled here
- editing of *commands*' sequences – equivalent to the solving of sub-problems
- points in time of learner's program executions (*play*) in order to test the (partial) correctness of his solution attempt
- system *error* messages
- chronological evaluation of the numbers of "actions" and "objects" mentioned above

What need not be recorded are interactions like "artistic" redesigning of the appearance of Kara's world, saving solution attempts etc.

3.2 Problem Solving Process

Each problem has an (unrequested) initial state and a (requested) goal state. The intermediate problem states build the problem space [10]. Consequently, problem solving means searching a correct path through the problem space from the initial state to the goal state. In the field of psychology the problem solving process is divided up into two phases, i.e.:

- building the problem space and
- searching the path through the problem space

Real problem solving does not require a complete problem space to be built before searching the correct path.

3.3 Problem Solving Strategies

If you start by building the complete problem space, you will find two different alternatives to continue:

hill climbing

It is a forward thinking strategy where the learner always tries to find an optimal solution for the respective next step. After controlling their success the learners continue to search for the next step. In case of unsatisfactory situations for Kara, they will have to do another (additional) modification to find the optimal solution for the step. Therefore their solution attempt is improved stepwise.

trial and error

The learner tries to find the correct way through the problem space (sometimes aimlessly) trying different possibilities one by one. This strategy is preferred for solving problems which seem complex or difficult from learners' point of view [4]. It is not possible to decide automatically whether the learner's errors are "good" or "silly" according to Edelmann.

If the problem space is divided into smaller sub-problems there are another two possible ways to continue:

top down

Using this strategy means to search *all* intermediate states before starting the solution, i.e. before finding the correct way through the problem space. This process employs the "divide and conquer"-idea of Informatics well known in software engineering.

bottom up

The learners solve every single sub-problem as soon as they identify it, even before they search for other sub-goals. The solution of the problem is completed after solving the "last" sub-problem. Exactly in this way this strategy is only applicable if the sub-goals are neither crossed nor nested.

3.4 Support for the Learners

Furthermore studies of several teachers in school as well as reports and discussions at faculty conferences and on-the-job trainings are regarded additionally to the examinations mentioned above. Based on this, individualized support was designed in preliminary considerations (see Table 1).

In the cases of good or very good quality of the learner's solution attempt the technical error message complemented with a special comment depending on the chosen problem solving strategy is used. We assume that these learners will understand the technical message. Due to different learners focuses the messages for bad or worst quality were designed. Although the trial and error strategy is not bad per se, it should however be changed to structured program solving if the solution's quality is medium or worse. Therefore the limit in the row for trial and error is at another point than the remaining.

These ideas for support have been designed in a way that can be realized for automated feedback by the learning environment, be it that the learner's problem solving strategy is identified.

Table 1. Individualized feedback after identifying the problem solving strategy

problem solving strategy	**Quality of the solution attempt**				
	very bad	**bad**	**medium**	**good**	**very good**
hill climbing	hint for structuring the problem first		technical error message help for the *actual* sequence		
trial and error	instruction for structured problem solving			technical error message help for the *incorrect* sequence	
top down	correct number of branches: hint to the *incorrect* branch		technical error message motivating comment concerning the well structured problem solution		
	wrong number of branches: claim the missing branch				
bottom up	correct number of branches: hint to the *incorrect* branch		technical error message help for the *incorrect* sequence		
	wrong number of branches: claim the missing branch				

3.5 Strategies' Choice Influencing Factors

An interesting question is, whether and why learners prefer a certain problem solving strategy to others. Possible causes for the learners' decisions – easily observable with the described research instruments – are:

- the time spent on problem solving
- the success in solving a task
- the task's difficulty

Additionally have to be considered the factors individual and lessons in school.

If a correlation between the used problem solving strategy and one of the first mentioned other factors is found, we will not have to

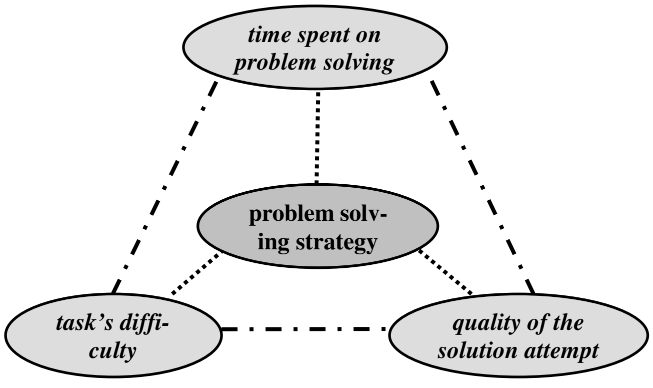

time spent on problem solving
problem solving strategy
task's difficulty
quality of the solution attempt

Figure 2. Correlations of influencing factors of the learners' choice of problem solving strategy

identify and categorize the way of proceeding with new methods, but rather use the easier measurable items time or quality.

Possible relevant correlations are shown in Figure 2. The data collected in the first studies (see Section 6.1) suggest the conclusion, that the quality of the solution attempt and the time spent on problem solving has only a little correlation. Between this time and the task's difficulty a positive correlation can be assumed. This assumption must be improved in further studies as well as the assumed negative correlation between the task's difficulty and the quality of the solution attempts.

Furthermore a medium correlation between problem solving strategy and quality of the solution attempt on the one hand and the time spent on problem solving on the other is noticeable. This correlation should be verified in further studies.

To be able to analyze the correlation between the (in the learner's opinion) difficulty of the task and the chosen problem solving strategy in further studies the test subjects have to fill in questionnaires additionally (see Section 5.3).

In the following field studies we want to see, whether there are correlations between chosen problem solving strategy and the remaining factors school lessons and individual. If the results in every participating class show the same distribution, we assume that there is no higher correlation between used strategy and school lessons. If additionally a certain test person shows the same strategy independent of the set task, can be assumed a high correlation between the chosen strategy and the individual factor.

4. PROCESS OBSERVATION METHODS

Previous systematic studies concerning programming novices' problem solving strategies were carried out predominantly at colleges and universities. In the work described here we tried to avoid the well known process observation difficulties described below. We have the well-founded hope, to generalize the results of the studies described here for learners in CS1 courses using different learning and programming environments than the one mentioned here.

Hundhausen describes in [7], how to analyze the programming process of students dependent on time using a given programming environment with the help of screen-video recording. At first semantic subunits of the given problem have to be determined based upon theoretical considerations. Afterwards the steps which should be identified in the subsequent recording process have to be coded manually – which takes a huge amount of time –, related to these categories and finally plotted against time. The problems detected by Hundhausen during his research as well as the difficulties described by Chi [2] during his analysis of "verbal" data made us design and develop specialized research software, which collects data automatically and presents it graphically. A similar method was chosen by Schulte [13]. In his case the test persons were pupils (11th graders, 16 to 17 years old), the topic was object-oriented modeling.

In order to achieve automatic categorization of the learners' ways of proceeding by a diagnostic software (see Section 5.2) another software tool will have to be developed the task of which is to identify patterns in the collected data of the tracking software (see Section 5.2) with the help of pattern-recognition methods.

Apart from the analysis of the problem solving process an analysis of the artifacts produced by the learners during this process is advisable. To evaluate the quality of the learners' solution attempts, these are tested with the help of test cases specially designed for this purpose. The test cases are selected so that an essential solution part is tested by each of them. In this way a more differentiated evaluation of the solution which does not only provide results like "completely correct" and "incorrect", is achieved.

5. RESEARCH METHODOLOGY

5.1 Research Objectives

Solving algorithmic problems often causes problems for programming novices. So one of the goals of the work described here is gaining more knowledge about the learners' individual strategy to construct a solution, which will finally improve the learning process.

Getting more knowledge about the learners' individual proceedings when constructing a solution with a view to enhance the learning process is another goal of the work described here. Therefore special research and diagnostic modules are added to the learning and programming environments used in the lessons.

In order to improve the learning environments the learners' different strategies must be identified and categorized automatically. Subsequently we want to associate these findings to the problem solving process.

With the system feedback messages improved, learners should be better able to solve arising problems alone by themselves, which should also augment their achievement motivation [8].

5.2 Research Instruments

In order to avoid the well known difficulties in process observation, specialized research instruments have been developed:

tracking software (TrackingKara)

- records the task setting
- records all steps relevant to solution
- categorizes the collected data
  - working with states
  - creating and editing branches
  - inserting and handling commands
- records the types of sensors and commands, the system error messages and the learners' reactions to these messages
- additionally records number and type of sensors, commands, system error messages, …
- takes snapshots of the learners' solution attempts to evaluate the quality of the solution process afterwards [6]

diagnostic software (EvalKara)

- allows the analysis of the collected data in shorter time also for larger groups of subjects
- supports the analysis of the development of all data over time
- supports graphical visualization of all data for further analyses by human researchers
- provides two kinds of cumulative analyses
  - time distribution
  - error distribution
- provides two different types of process diagrams
  - activity-time diagram

- element-time diagram
- supports the evaluation of the quality of learners' solution attempts with the help of test cases specially designed for this purpose

5.3 Research Process

Software requirements for these software-based research instruments were derived from theoretical preliminary considerations and from several test scenarios accomplished with four test persons (different background in computer science: two novices, a student of computer science and a former programmer) in the spring of 2007. Each test subject was given one or more typical Kara tasks (see Section 2). During the problem solving process they were observed and interviewed afterwards. The research instruments were designed and developed according to the requirements described above.

In a first case study 10 test persons with different pre-knowledge levels in informatics (scaling from beginners up to students of informatics) were asked to solve the following three Kara tasks:

- A: (*Kara and the leaves*) Kara has to invert a pattern of cloverleaves straight in front of it while moving towards a tree trunk, in front of which it has to stop (see Figure 1).
- B: (*searching the tunnel I*) Kara has to find the entry of a "tunnel" formed by tree trunks straight in front of it and to stop there.
- C: (*searching the tunnel II*) Kara has to find the entry of a "tunnel" formed by tree trunks straight in front of it and to stop at the end of the "tunnel".

Their solution steps were recorded automatically using the tracking software. Additionally the test subjects were asked to comment their method of proceeding by the so called "thinking aloud" method, which should ensure the correct interpretation of the collected data. Their statements were recorded by a researcher and led to first rules necessary for the interpretation of the data collected by the tracking software. On this basis, further requirements for the diagnostic software were derived and finally integrated.

After the first revision of the software-based research instruments first studies with larger groups of test subjects were carried out. About 100 pupils (7th grade, 12 to 13 years old) of two Bavarian grammar schools took part in these studies (approved by the Bavarian State Ministry of Education and Religious Affairs). In Bavaria Informatics is compulsory for all learners in the 6th and the 7th grade (1 lesson per week). In the 7th grade the curriculum requires the description of sequences with algorithms. Before the learners are able to analyze and construct such presentations on their own, they have to learn about the basic control structures such as sequence, choice and loop (using the learning and programming environment Kara in this case) in about 8 lessons. Because of the given limitations it could be assumed that the learners' level of pre-knowledge were similar so far. During the studies the learners were asked to solve the three tasks mentioned above individually (one pupil per computer) within a period of 45 minutes. The subjects, however, were allowed to communicate.

What should be found in further field studies assisted by questionnaires is a fine categorization of different problem solving strategies, which could be detected by the new diagnostic software. Furthermore we wanted to know to what extend the system error messages are helpful to the learners.

6. FIRST RESULTS

6.1 General Statistical Results

With the help of the collected data and the snapshots of the learners problem solving taken by TrackingKara a retrospective examination of the problem solving process the following results were realized. Figure 3 to Figure 5 show boxplots for task A and task B. Each boxplot represents the five-number summary (minimum – smallest observation, lower or first quartile – which cuts off the lowest 25% of the data, median – middle value, the upper or third quartile – which cuts off the highest 25% of the data, maximum – largest observation) of a data set in descriptive statistics. The vertical line shows in each case the maximum and the minimum value. The box displays the lower quartile and the upper quartile. The horizontal bar indicates the median. The mean value is not shown explicitly in the diagrams.

According to the results mentioned above the average amount of time spent on solving is 255 seconds, which is due to a very high average amount of time for task C (more than 400s) and two lower values for task A and B, which show similar results (see Figure 3 – left). One reason for this may be the fact that the complexity of task A is similar to that of task B.

The average number of system error messages during the solving of one task is 1.6 no matter if you regard task C or not. What is striking here is a different average in the number of system error messages between task A and task B (see Figure 3 – right) despite to their similar level of difficulty. The average number of system error messages decreases from 2.2 to 0.8. The difference might be explained by practice effects, an assumption which should be verified in further studies supported by questionnaires.

The average time elapsed till the first system error message comes up is approximately two minutes – thus about one third of the complete solving time (see Figure 4). Therefore it seems to make sense, to create the individualized feedback messages for the learners, which can be helpful to their further steps.

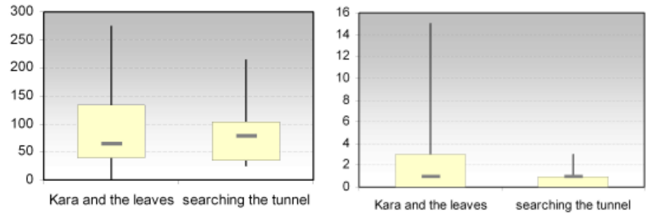


Figure 3. Time spent on problem solving in s (left) – number of system error messages per solution (right)

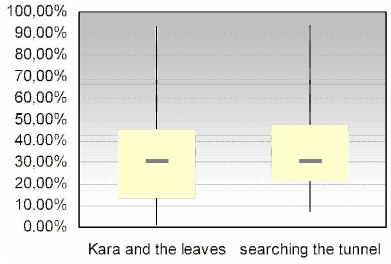


Figure 4. Time until first system error message occurred in percent of the complete solving time

The quality of the learners' solution attempt was evaluated by two Informatics teachers. In the German grading system (from 1 to 6 – with 1 being the best and 6 the worst achievement) the average mark for task A is 3.0, the one for task B 2.27 (see Figure 5). This also may be a sign for practice effects, which must be verified in further studies, too. In addition to that consistency of the teachers' assessment and the results of the test cases of EvalKara (see Section 6.3) must be achieved as far as possible.

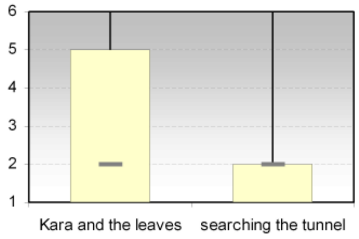


Figure 5. Quality of the solution attempts (German marking system)

Another point of interest is the correlation between problem solving strategy, quality of the solution and the time spent on solving (see Figure 2 in Section 3.5). The first results show medium correlation between the problem solving strategy and the solving quality on the one hand and the spent time on the other, whereas there is only very little correlation between the quality of the solution and the time consumed by solving. This fact is already well known in the homework research in mathematics and other school subjects.

6.2 Analyses of Individuals

Based on the classification in categories of the collected data described in section 3.1 and their chronology during the problem solving process the strategies listed in 3.2 should be identified automatically. EvalKara's so called *activity-time diagrams* (see Figure 7 to Figure 10) provide assistance for the analysis. They show the distribution of a test subject's categorized activities in comparison to time. Certain combinations of reported learner-system-interactions lead to a classification of the data into four groups of "strategy-patterns" (see Figure 6).

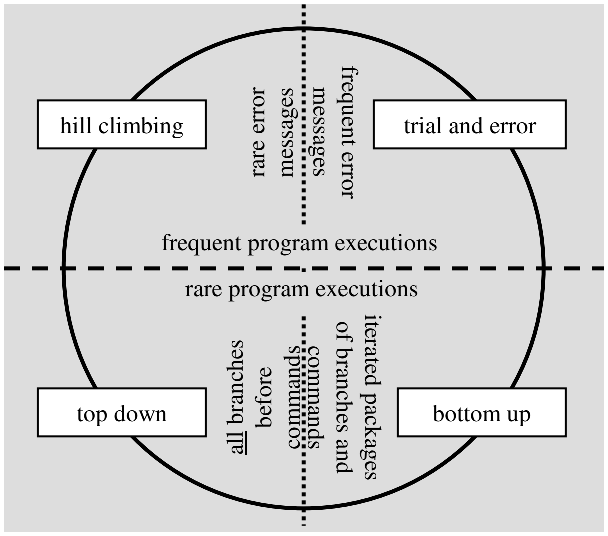


Figure 6. Identification of four different patterns

If the learner creates only one branch (mark at transition/branch) at first and then edits the conditions before completing this branch with a sequence of commands, the "bottom up"-pattern is attributed. Repeating the same order does not change this classification, additionally any other *single* interactions (e. g. program execution) during the sequence of actions are ignored. However, multiple subsequent interactions cause a change of the pattern attribution or at least the transition to the initial setting "no identifiable pattern".

If creating further branches follows the creation of the first branch directly and subsequently *all* related conditions are built before at least *all* commandos are inserted, the "top down"-pattern is attributed. In this case the attribution of system-learner-interactions' chronology and strategy-pattern has a tolerance of one, within the number of branches can differ from the correct value.

Similar rules for attribution were defined for all the other strategies. Based on the graphical presentation of the collected data in process diagrams provided from EvalKara the learners' individual problem solving strategies were analyzed by human researchers. In the studies (200 reported sessions) described here the attribution of strategies and patterns was accomplished by two observers. They agreed about the attribution of the four different strategies (see Section 3.3) to the according patterns.

The diagrams in Figure 7 show examples of a problem solving process, where at first the test subject creates all necessary states, afterwards all branches (and transitions) and at last he/she fills in the respective commands in every branch (see mark in Figure 7). He/She divides up the problem space in smaller sub-problems which are not solved until the building of the problem space is completed. This is in accord to the top down problem solving strategy described in Section 3.3.

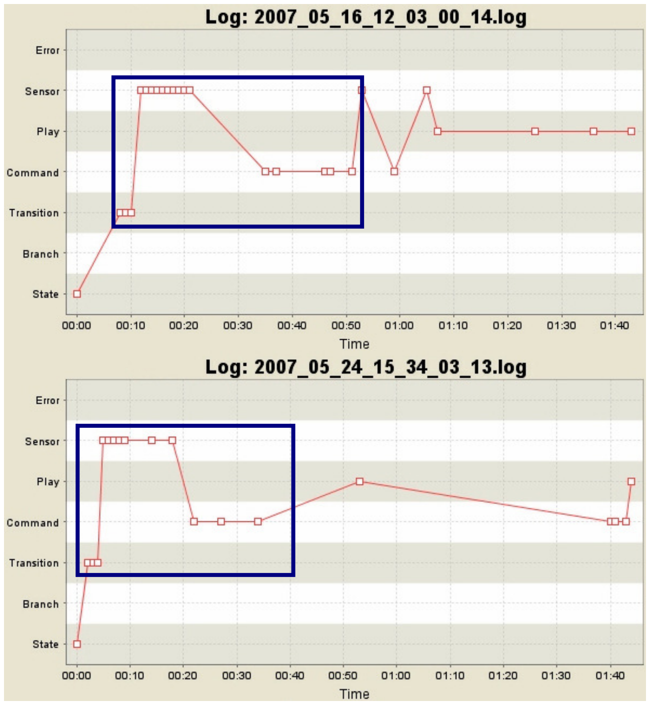


Figure 7. Activity-time-diagrams – presentation of the top down problem solving strategy

Before finishing the solution eventually the test person accomplishes several improvements.

The diagrams of Figure 8 again show a problem solving strategy, where the complete problem space has at first been divided up into smaller sub-problems. Here, however, the commands are filled in a branch just before the next branch has been created and edited (see mark in Figure 8). The final solving of the complete problem results from the solving of the sub-problems as soon as the editing of the last branch is completed. This way of proceeding accords to the bottom up method described in Section 3.3.

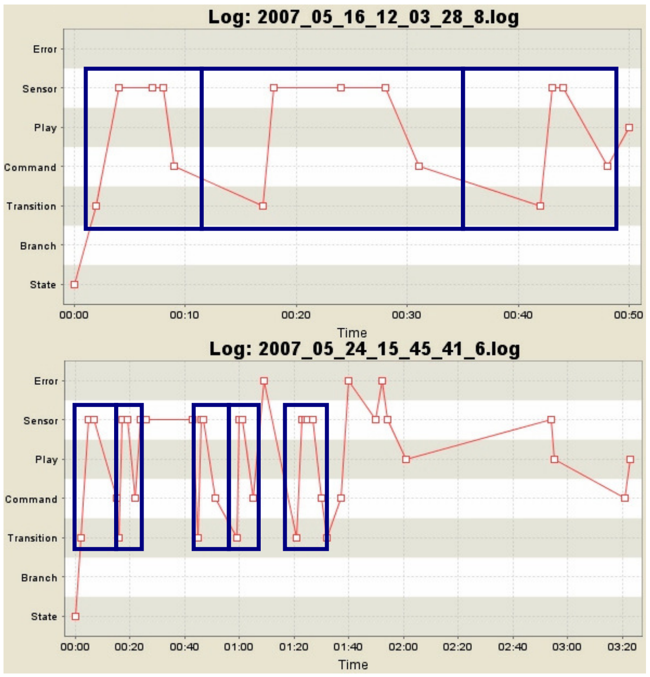


Figure 8. Activity-time-diagrams – presentation of the bottom up problem solving strategy

The repetition of the marked pattern in Figure 8 displays the consecutive finishing of the several sub-problem's solving. The learner-system-interactions between the patterns help to improve the solution of the respective branch.

Some learners use a pure trial and error method: each step of the solution is tested by immediate program execution (see marked selection in Figure 9). Furthermore it can be assumed, that the

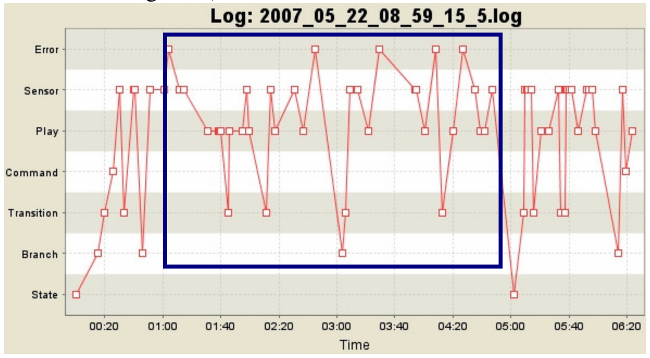


Figure 9. Activity-time-diagram – presentation of the trial and error problem solving strategy

occurring system error messages affect the learners' next steps. If you want to analyze this in detail, you must consider the types of error-messages. The graphical representation of this problem solving strategy clearly differs from the ones in Figure 7 and 8.

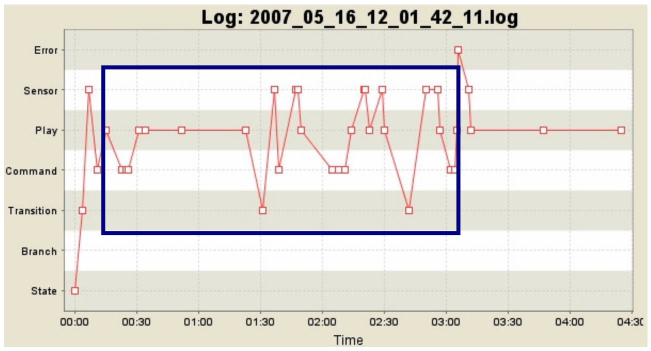


Figure 10. Activity-time-diagram – presentation of the hill climbing problem solving strategy

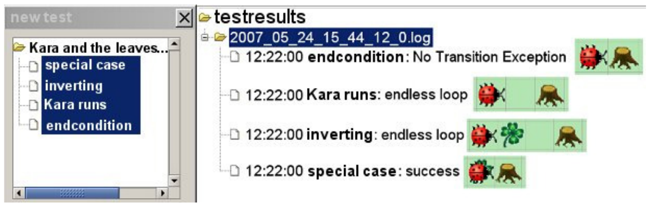
In addition to this you cannot only identify the hill-climbing strategy (see Figure 10) described in Section 3.3, where there are no system error messages in between the program execution, but where the learner himself assesses Kara's situation as disappointing and improves his solution attempt step by step.

Besides, compositions of these four classes of problem solving strategies can be found in terms of different patterns in the activity-time-diagrams.

Another result is, that learners, who have started the problem solving by a certain strategy for a new start often still employ the same problem solving strategy after the occurrence of difficulties. An exception are test subjects, who change their problem solving strategy from a structured one to the trial and error method, because system error messages came up repeatedly and thus they failed in finding a correct solution.

6.3 Analyses of the Results of the Test Cases

EvalKara provides a tool based on test cases for the quality assessment of the solution attempts. Each of these test cases was especially developed to check up one essential concept of the concerning solution, for example when task A is tested, the system checks if Kara runs, stops at a tree, inverts the pattern of leaves and if it executes a special case (see Figure 11) successfully.



2007_05_24_15_44_12_0.log
12:22:00 **endcondition**: No Transition Exception
12:22:00 **Kara runs**: endless loop
12:22:00 **inverting**: endless loop
12:22:00 **special case**: success

Figure 11. Example for the results of test cases (task A)

The results of the test cases achieved by EvalKara are "success", "endless loop" or the according system error message. The combination of these results gives a first notion of the quality of the learner's solution attempt. Table 2 is an overview which shows how to “transform” the combinations of the test case results into marks (German school marking system – mentioned above in Section 6.1).

Table 2: attribution of combinations of test case results (inaccessible ones ignored) and marks

test case	success				mark
endcondition	yes				**1**
Kara runs	yes				
inverting	yes				
special case	yes				
endcondition	yes				**2**
Kara runs	yes				
inverting	yes				
special case	no				
endcondition	no	yes			**3**
Kara runs	yes	no			
inverting	yes	yes			
special case	yes	yes			
endcondition	yes	no	yes		**4**
Kara runs	yes	yes	no		
inverting	no	yes	yes		
special case	no	no	no		
endcondition	no	yes	no	no	**5**
Kara runs	no	no	yes	no	
inverting	yes	no	no	yes	
special case	yes	no	no	no	
endcondition	no				**6**
Kara runs	no				
inverting	no				
special case	no				

A learner, whose solution attempt produces the test case result “success” only for the test cases “endcondition” and “Kara runs”, gets mark 4. The automatic marking of EvalKara was compared with the marks two Informatics teachers gave for the same solution attempts (see Section 6.1). At the moment remaining differences between these assessments are used to improve the composition of the test cases and the attribution of combinations of test case results and marks (see Table 2).

7. CONCLUSION AND RESULTS

The results of the case studies presented here show that it is possible to identify different problem solving strategies by means of these newly developed research instruments. Therefore the data collected has to be scanned thoroughly. Methods of descriptive statistics are useful when the tasks difficulty (in the learners' opinion), the amount of time consumed, the quality of the solution and the respective solution process are tested concerning their correlation.

In further studies the next step (additionally supported by questionnaires) will be an in-depth analysis of the data collected (approx. 200 sessions) to approve the categories of problem solving strategies and the attribution of learners' strategies and patterns in the data. The questionnaires will be draft based on the ideas of Ajzen's Theory of Reasoned Action [1]. Afterwards, considering the results of the new studies, the categories will be refined.

The basic ideas of TrackingKara and EvalKara should be transferred to other common learning environments used in schools like e. g. Karel, the robot, to develop similar software instruments. It must be clarified in the context of didactically reduced imperative programming languages what the relevant learner-system-interactions are – which have to be recorded. What is also important is to test the effectivity of the currently existing system error messages with the help of additional questions in the questionnaires mentioned above, which could be realized in further studies.

8. REFERENCES

[1] Fishbein, M., Ajzen, I. 1975. Belief, attitude, intention, and behavior: An introduction to theory and research. Reading, MA: Addison-Wesley. URL: http://people.umass.edu/aizen/f&a1975.html

[2] Chi, M. T. H. 1997. Quantifying Qualitative Analyses of Verbal Data: A Practical Guide. The Journal of the Learning Sciences, 6, 3 (1997): 271-315.

[3] Conway, M. J. 1997: Alice: Easy-to-Learn 3D Scripting for Novices. Doctoral Thesis. University of Virginia, School of Engineering and Applied Science.

[4] Edelmann, W. 1979. Einführung in die Lernpsychologie. Kösel, München, Germany.

[5] Hartmann, W., Nievergelt, J., Reichert, R. 2001. Kara, finite state machines, and the case for programming as part of general education. In Proceedings of the IEEE 2001 Symposium on Human Centric Computing Languages and Environments (Stresa, Italy, September 05-07, 2001). HCC'01. ACM Press, New York, NY, 135-141. DOI= http://doi.ieeecomputersociety.org/10.1109/HCC.2001.995251.

[6] Higgins, C., Symeonidis, P., Tsintsifas, A. 2002. The marking system for CourseMaster. In Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education. ITiCSE '02. ACM Press, New York, NY, 46-50. DOI= http://doi.acm.org/10.1145/544414.544431

[7] Hundhausen, C. D. 2006. A Methodology for Analyzing the Temporal Evolution of Novice Programs Based on Semantic Components. In Proceedings of the 2006 International Workshop on Computing Education Research. (University of Kent, Canterbury, UK, September 9-10, 2006) ICER '06. ACM Press, New York, NY, 59-71.

[8] Kiesmüller, U.; Brinda, T. 2008. How Do 7th Graders Solve Algorithmic Problems? – A Tool-Based Analysis. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (Madrid, Spain, June 30-July 2, 2008). ITICSE 2008. ACM Press, New York, NY, 353.

[9] Maloney J., Burd, L., Kafai, Y., Rusk, N., Silverman B., Resnick, M. 2004. Scratch: A Sneak Preview. In Second International Conference on Creating, Connecting and Collaborating through Computing. (Keihanna-Plaza, Kyoto, Japan ,January 29-30, 2004) C5'04. IEEE Computer Society, Los Alamitos, CA, 104-109. DOI= http://doi.ieeecomputersociety.org/10.1109/C5.2004.1314376.

[10] Mayer, R. E. 1992. Thinking, problem solving, cognition (2nd edition). W. H. Freeman and Company, New York, NY.

[11] Pattis, R. E. 1994. Karel The Robot: A Gentle Introduction to the Art of Programming, 2nd Edition. John Wiley & Sons, Inc., New York, NY.

[12] Reichert, R. 2003. Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science. Doctoral Thesis. No. 15035. ETH Zurich. URL: http://e-collection.ethbib.ethz.ch/show?type=diss&nr=15035

[13] Schulte, C. 2004. Empirical Studies as a tool to improve teaching concepts. In Informatics and student assessment. Concepts of Empirical Research and Standardisation of Measurement in the Area of Didactics of Informatics. Magenheim, J., Schubert, S. (eds.). Köllen, Bonn, Germany. 135-144.

[14] Schwill, A. 1997. Computer science education based on fundamental ideas. In Information Technology – Supporting change through teacher education. Passey D., Samways B., (eds.). Chapman Hall, London. 285-291.

Understanding TDD in Academic Environment: Experiences from Two Experiments

Sami Kollanus [*]
Department of Computer Science and Information Systems, University of Jyväskylä
Finland
sami.kollanus@jyu.fi

Ville Isomöttönen[†]
Department of Mathematical Information Technology, University of Jyväskylä
Finland
vilisom@jyu.fi

ABSTRACT

Several studies have reported positive experiences with Test-Driven Development (TDD) but the results still diverge. In this study we aim to improve understanding on TDD in educational context. We conducted two experiments on TDD in a master's level university course. The research setting was slightly changed in the second experiment and this paper focuses on comparing the differences between the two rounds. We analyzed the students' perceptions and the difficulties they faced with TDD. The given assignment clearly affected the students' reflections so that the more difficult assignment evoked a richer discussion among the students. Additionally, some insights into teaching TDD are discussed.

Categories and Subject Descriptors

K.3.2. [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Experimentation

Keywords

TDD, Education

1. INTRODUCTION

Beck published his first edition on Extreme Programming (XP) [1] almost a decade ago. TDD has become one of the best known and possibly the most widely used practice of XP. The basic idea of the practice is simply to write tests before the production code. This is done iteratively in very short, even minutes long cycles.

[*]Arranged the experiment, conducted the analysis, and provided the conclusions.
[†]Conducted the analysis, and provided the conclusions.

In several studies, TDD has been suggested to improve code quality [3] [14] or productivity [4] [6]. Also some other benefits, like improved reuse [11], have been reported. However, the evidence on these benefits is not that clear. The existing results are contradictory and more research is needed to better understand the essence of TDD and its benefits for the software engineering field [5].

Many educators have followed and applied XP's testing/-design practice, TDD, in varying stages of CS/SE curricula. Several positive experiences have been reported [2] [9] [13], but is there actually any agreement among the academics on what it is, what it achieves, what it achieves in the CS/SE education, and then, how it should be embedded in the curricula. Some educators have proposed that it should be embedded in the very beginning of the curricula [2] [9]. However, it may not be so straightforward to have it at CS1 level. For example, Keefe et al. [7] reported that TDD was the most difficult of all the XP practices for their students. Also we concluded in our previous study that learning TDD caused high cognitive load even for the advanced students [8]. The major question is what is regarded as learning objectives at varying levels of the curricula.

The general long-term aim of our work is to gain better understanding on the role of TDD in CS/SE education. With this objective, the first and very difficult task is to illuminate what TDD actually is and achieves. We consider that previous research has not been able to sufficiently explicate this basic question.

Our current work aims to study the phenomenon from the students' point of view. We started this in our recent study which focuses on the students' perceptions on TDD [8]. We conducted an experiment in the master's level course on testing and quality assurance at the University of Jyväskylä. In order to better trust the conclusions derived from the students' experiences we replicated the original study with minor changes. In this paper we report results from these two experiments including a comparison between them.

We start with the description of research setting and process in Section 2. In Section 3, we present the background and the results of the first experiment. This is followed by the corresponding presentation on the second experiment in Section 4, in which we also analyze the differences between the experiments. We discuss the results and summarize the main conclusions in Sections 5 and 6.

2. RESEARCH SETTING AND PROCESS

The context in our study was a university course on software testing and quality assurance. We conducted two experiments, in autumn 2006 and in spring 2008. The participants in the course were master students, mostly majored in software engineering. The course is planned to be a fourth year course, but actually the students had varying background because of flexibility in our education system. Some of the participants were pretty young third year students, but many of the students had already several years of working experience. The minimum prerequisites for the course were two basic programming courses and two other courses related to software engineering methods and processes. However, most of the students had much better programming skills than the basic courses require. TDD was relatively new approach to the students. Some of them had tried it before, typically on another course. Only few of them had used it in work.

The research process aims to increase understanding of what TDD in CS/SE education actually means. We have started this work by analyzing the students' experiences without particularly focusing on any tight research question. We are interested in how the students perceive the method, what it actual means, how it should be taught, and where in the curriculum it should be taught?

The data originates from a questionnaire applied in the both experiments. Based on this relatively small data we are mainly interested in what perspectives emerge in the students' experiences, without an objective to claim relations between the emerged concepts. Thus, the data allows an initial coding under the theme "TDD in CS/SE education", and the aim is to analyze what hypotheses rise for a further conceptualization.

An important point to disclose about the research process is that the hypothesis raised by the first experiment guided the selection of programming assignment in the latter; We noticed that the difficult programming assignment in the first experiment hindered a bit the students' concentration purely on TDD. In the second experiment round, we tried to make the concentration easier with a simpler programming assignment.

2.1 Structure of experiments

The both experiments included the following steps:

- introduction lecture
- little "warm up" assignment
- actual programming assignment
- questionnaire

First, TDD was briefed only in one introduction lecture (90 min.). Simple "by the book" introduction was presented with very small examples and without academic debate. In addition, the students were advised to individually go through the bowling game kata example [10]. TDD was very little discussed during the rest of the course. So, the students formed their conceptions pretty much individually based on their experience with the coursework.

The programming assignment included two tasks. The first part was very easy warming up with TDD and the tools, which were Eclipse with JUnit. The primary programming assignment was different in each experiment round. In each case the assignment aimed to be challenging enough without requiring too much time. Support sessions (3*2 h) were available for the students who needed it.

Finally, the students had to reflect and report their experiences on TDD. We collected the experiences with a questionnaire, which is presented in appendix A.

This assignment including the questionnaire was compulsory for the students to pass the course. Therefore the participants represent a wide spectrum of students, not only the most motivated ones. Some arrangements were different in each experiment round. These differences are explained in the following sections.

2.2 Data analysis

We followed the coding guidelines of Grounded Theory [16] as a rigorous approach to our data analysis. We first applied open coding to discover relevant points in the data, which was followed by identifying causalities. With the small data the causalities are still relatively weak but provide basis for the future work. The contribution of the paper is not a theory generation, but it could be a start for conceptualizing the area of TDD in education.

The data was managed according to the following steps. First, the raw data was transfered into a table. The answers were relatively short, so we were able to visually manage each answer in a single row. Second, we picked up the relevant issues in each answer, and wrote them down to the second column of the table. Third, through constant comparison, the issues were conceptualized into categories. Additionally, some notes were written down during the whole coding process. The notes mostly consisted of possible causes for the key findings. Finally, we aimed to find causalities by analyzing the found categories against the raw data and the written notes.

Both authors conducted the analysis individually, after which the results were compared and discussed for an agreement. The data from the both experiment rounds was first handled separately, which was followed by a comparison between the experiments. This final comparison was made as pair work.

3. FIRST EXPERIMENT ROUND

The first experiment was conducted during the autumn 2006. The students had to do the TDD assignment as a part of their coursework, which formed 25 % of the course grade. They were strongly recommended but not forced to work as pairs. As a result we received 27 completed assignments and responses to the questionnaire from 25 pairs and two individuals. So, the total number of the participants was 52.

The primary task in the assignment was to implement a simple HTTP server. This task aimed at setting enough challenge for the participants to try TDD with realistic cognitive load in programming work. The compulsory courses in our curricula don't include all the knowledge needed in the assignment, but support was available for those who needed it.

3.1 Students' Background

TDD was relatively new approach to the students. Only in one pair both of the students had tried TDD before the course. One of a pair had tried it in 10 pairs and the rest 16 (14 pairs and two individuals) didn't have any earlier experience. So, 11 of the 27 participants (pairs or individuals) had some earlier experience with TDD. Their experience was mostly pretty lightweight and related to a brief introduction on another university course. Only a couple of the students had experience with TDD in real programming work.

Three of the pairs reported they had followed TDD very strictly and also three of them had slipped often. Most of the respondents (21) reported they had slipped from the test first principle few times or occasionally. We concluded that the students were motivated and seriously tried to follow TDD despite of the difficulties.

Most of the pairs (15) did at least part of the assignment together using the same computer (answers 1 or 3, see appendix). The rest of the students worked mostly individually.

3.2 Experiences

3.2.1 How hard was it to use TDD (scale 1-5, see appendix)?

The experiences were quite equally divided to easy and hard side of the scale. The most typical answers were 2 (9 answers) and 4 (10). The average of the answers was 3.0 and also median was 3. We also analyzed how earlier experience with TDD or work organization affected this answer. The result related to students' earlier experience on TDD was pretty obvious. The students without earlier experience had more difficulties with TDD. However, the difference between the groups was not huge. This is possibly because of very limited earlier experience even among the students who had tried TDD before.

Work organization made more clear difference between the answers and this was not so obvious for us. The students, who worked together, had more difficulties with TDD. Could pair working lead to some difficulties in this kind of assignment? We don't know the answer, but we considered that likely the more skilled students preferred individual work.

3.2.2 Would you like to use TDD after this experience?

This was an open-ended question, which aimed to get information about the students' attitude towards TDD after the assignment. We analyzed the textual answers which divided into three categories. The attitude of the students was more positive than we expected. 20 of the 27 pairs were more or less willing to use TDD in the future. Three of them answered neutrally and only four pairs would not like to use TDD in the future.

The most interesting finding for us was to see, how many of the students believe the expected benefits of TDD regardless of their difficulties with the assignment. Many of the students wrote that TDD improves their code quality and they can trust better on their own code. They saw TDD as an extra cost in their work, but they still believed more in expected benefits. However, several pairs claimed that TDD is not applicable in all projects and they would need much more training to be able to use it effectively. So, the students simply appeared to believe in the claimed benefits, even if they didn't really experience them in this assignment.

3.2.3 What was most difficult in the assignment?

The answers to the question divided into the following categories:

- implementation
- writing of appropriate tests
- abstraction level
- discipline/attitude
- other

Implementation. The implementation of the server turned out difficult for the students. The majority of the participants quite clearly reported implementation difficulties as they were typically stuck in a small-scale technical issue. Many of the implementation issues were actually related to the test code, not the server itself. Writing the test for web server in JUnit environment set unexpected technical challenge for the students. Several of them became frustrated because the issues with the test code took most of their time. The students also reported problems due to poor programming skills, new programming language, or new tools. This category does not directly relate to TDD but is a reason for the other perceived difficulties.

Writing of appropriate tests. The category means that the students did not know what kind of tests should be written. The students' considerations include the extent of a test, i.e. the number of tests in single test. We consider that the extent of a test becomes easily interpreted "not by-the-book", so that every possible aspect and future extension should be included in a test. In fact, this has been called exhaustive testing [15]. Another interesting detail is the difficulty to implement a test with JUnit's `assertequals` expression, which means the difficulty to approach TDD with the chosen tools. It seems that the novelty of the test-first paradigm requires explicitness in teaching in order to show how a test looks like, what it is, what it strives for, and how it is written (tools). The introductory lecture, provided in the course, underlined the issues but did not provide a thorough hands-on demonstration.

Abstraction level and TDD. The answers indicated that it was hard to test the server. We placed the answers into the category of abstraction level for the following reason: the task dealt with integration (server-client) level, which introduced difficulties in approaching TDD. For example, the students couldn't proceed in small steps. Rather, they perceived the client side as a test and forgot that also the actual implementation of the server could have been implemented with TDD. Interestingly, the students seem to wonder how the method scales up, and also question why the trivial functionalities should be tested. We conclude that despite the programming difficulties in general, the chosen assignment revealed serious questions here.

Discipline. In the answers of six pairs the discipline was brought up. One of the answers relates to the discipline for writing a pretty code. The others are more relevant, bringing up the difficulty of following of the strict method. Some explain this by emphasizing the novelty of the test-first model, and some couldn't get rid of their old habits when learning the model. One pair explicitly notes that it probably wouldn't be so difficult if the model was taught as the first programming model.

Other. An issue that was occasionally reported as a reason for the other categories was the novelty of the test-first paradigm. We suppose that the issue is a general expression for the writing of appropriate tests. Also, it seems to relate to the disadvantage of old habits. The data revealed also other aspects without clear reoccurrences. One of them was the straggling of the code, which we interpret to be due the lack of any initial design, resulting in the code that was perceived as uncontrolled. Obviously, the bottom-up approach (TDD) cannot omit a sufficient initial understanding of the problem domain as a whole. Yet, one pair comments on the initial step of TDD, i.e., the guideline to see the negative test result at the start, when there's not any of the actual production code available. The students couldn't figure out the purpose of the action and perceived it as an unnecessary hype.

4. SECOND EXPERIMENT ROUND

The second experiment was conducted in spring 2008. We did some changes to the research setting after the first round. The most important change was the programming task. The students had many technical difficulties with the task and tools during the first round. Therefore we gave the students an easier task to enable them be better focused on TDD. The students had to do a small software component, which can be used to store, retrieve and parse certain kind of text.

We had 27 participants in the experiment. The assignment was again compulsory part of the course, but this time it didn't have effect on the course grades. Possibly because of the course arrangements, the students didn't appear to be so eager to pair work. We received 20 completed assignments from 7 pairs and 13 individuals. We considered the interpretation of the answers of the pairs little problematic in the first round, and this is why we asked all the students to individually fill the questionnaire in the second round.

4.1 Students' background

We asked the students about their working experience on ICT field. Most of the students worked at the same time with the course. Only 8 of the 27 students didn't have any working experience. The experience of rest 19 (70 %) students varied from 6 months to 12 years (md = 26 months). In the first round we didn't ask this, but these results confirm our estimation about the students' background. They are really experienced compared to most of the student experiments in software engineering research.

TDD was again quite new approach to the students. Almost half of them (13) had tried TDD before, but most of them had done it only on another university course. Only one had used it systematically in work.

Most of the students clearly tried to follow TDD rigorously. Only one reported that he wrote test systematically after code and three of them had slipped very often from the test first principle. Majority of the students had slipped occasionally or only few times. So, they appeared to be motivated to try TDD seriously. On the other hand, only three students reported that they didn't slip at all from the test first principle. This obviously relates to the difficulties in maintaining discipline.

4.2 Experiences

4.2.1 How hard was it to use TDD

The easier programming task affected the students' experiences on TDD. Average of the answers was 2.6 (first round 3.0), which means that TDD was experienced as easier than in the first experiment. This was quite expected result, because an easy programming task leaves more cognitive capacity for learning TDD.

The earlier experience on TDD had only a small effect on the results, just like in the first experiment. TDD was slightly easier for the students with earlier experience. In the first experiment, we found that the students working in pairs had more difficulties with TDD. The second round produced opposite results. The students without a pair experienced TDD as more difficult, but this time the result was not so clear. We don't have a good explanation for difference between the experiment rounds. For some reason, most of the skilled students seemed to prefer working individually in the first experiment and working in pairs in the second round.

In the second round we were also able to analyze how working experience affected the students' answers. TDD was slightly easier for the students with working experience, but the difference was relatively small. Amount of working experience didn't have a remarkable effect on the results.

4.2.2 Would you like to use TDD after this experience?

Also after the second round, the students' attitudes were more positive than we expected. Only 4 of 27 students wouldn't like to use TDD in the future. In addition, five students were neutral in their opinions. This confirmed our findings from the first experiment. In this kind on small assignment the students don't have an opportunity to personally experience the claimed benefits of TDD. Many of them also found it difficult to follow TDD principles. However, they seem to be eager to believe in the benefits and would like use TDD in the future.

The most common reason for the positive attitudes was better trust in the code. TDD was seen as a safety net for the programmer. Some students also liked the approach, because it drives to small steps in the programming work. In this round, very few students considered test code quality or possible difficulties with refactoring. The negative comments focused on extra work needed in TDD practice or changing the routines they were used to. Some students just didn't like TDD, because they felt it disturbed their thinking process in the programming work.

4.2.3 What was most difficult in the assignment?

When asked what was most difficult in using TDD, the students raised a lot of same issues than in the first experiment. Three common categories of difficulties came clearly up. First, for many of the students, it was simply difficult to adapt to the test first approach. Second, several students reported it was hard to design proper tests, especially in very short cycles required in TDD. Third, many of the students suffered from technical difficulties. Some of them had not done programming in several years and for few of them this was the first experience with Java. Also some difficulties with JUnit environment were encountered.

The second experiment also disclosed new interesting dimensions. These include the following:

- TDD was perceived as a design method
- TDD was perceived as motivating
- there was a contradiction between the disciplined method and creativity

- the students experienced TDD applicable for pair work

Some students experienced that the program was designed along the usage of TDD. There was a difference compared to the first experiment with the more difficult programming assignment, in which some students felt their code uncontrolled, and we concluded that the this indicated lack of initial design.

With the easier programming assignment, motivation was also found. In fact, we expected this issue to come up already in the first experiment. The conclusion is that the short intervals of the method probably decrease the cognitive load (complexity) giving motivating feedback for the developer. In the first experiment, they had more difficulties to work in baby steps as they perceived that the client is the test in the http-server assignment.

Maybe the most interesting are the last two perspectives. The disciplined method was perceived as offensive because it disturbs a creative flow. This indicates that the relation between creativity and the disciplined method is unclear, and should be further studied. TDD was also perceived as applicable for pair work. TDD as pair work was suggested to make the test writing easier.

Overall, we noticed that the easier assignment in the second experiment did not raise a rich discussion among the students compared to the first experiment. The second experiment gave rise to well known reasoning of TDD, and in this sense, the answers seem to be closer to the by-the-book briefing lecture. In the second round, some respondents explicitly comment that TDD cannot be internalized with the easy assignment.

5. DISCUSSION

We noticed that realism played a significant role in how TDD was perceived. The more difficult assignment in the first experiment represented more realistic cognitive load in programming work. This raised rich discussion. In the second experiment, the students noted that it was difficult to internalize TDD, because the assignment was so simple. The both assignments disclosed that the students would have needed more examples on how to apply the method in practice. So, TDD should be first taught by showing how to do it. After being able to follow the method in practice, realistic assignments would increase critical thinking and recognition of yet unclear meanings (cf. self-direction).

The teaching approach on TDD should be considered based on the learning objectives. In our master's level course, the goal was to create overall understanding on TDD as a software engineering practice and to evoke individual thinking. With this goal the first experiment was more successful. The approach should be different if the objective was to give a practical hands-on training on TDD. This would require much more time, concrete examples, and step-by-step training.

These experiments already raise a discussion on what TDD actually is and achieves. It has been claimed to be a design method. With the easier assignment, some students experienced that TDD was a means to design, but with this simple assignment the design was easy to keep in mind. Interestingly, the more difficult assignment did not give rise to the design property of the method. Some students reported they felt uncontrolled with their code. We concluded that with increasing complexity the students' would have needed some up-front design, because the structure was difficult to keep in mind.

An important question is whether the test is written before the production code. Is the test writing prior to the production code only a way to document the design? This as a disciplined rule may be questionable in the unit level. For example, some students felt the contradiction with creative flow and the disciplined method. Based on the viewpoints discussed above, the meaning of TDD as design method should be further studied. Is the regression testing finally the only major benefit of the method. These are somewhat difficult questions which will direct our future work.

Müller and Höfer [12] reported that student experiments with TDD may not be widely generalizable. The strength of our study is that the students in our experiments were really experienced related to the most of the student experiments on the research field. Many of them were already professionals with several years of working experience. However, there are some restrictions in our study. First, the assignments were relatively small and they do not totally correspond to the reality of software engineering work. Second, even the most experienced students were novices with TDD. In order to understand TDD better we should direct the future research to overcome these challenges.

6. CONCLUSIONS

In this paper we have discussed experiences with TDD in educational context and reported the results from two experiments in university courses. The main conclusions are related to the difficulties the students had with TDD. These difficulties gave rise to three main categories:

- TDD approach
- Designing of tests
- Technical difficulties

Table 1 includes explanations for these categories. Additionally, some exemplars of the students' comments related to each category are provided.

The comparison between the experiments yielded three other interesting viewpoints. First, the experiences varied depending on the programming assignment. The more difficult assignment in the first round evoked a richer discussion, but the students with weak technical skills couldn't concentrate on learning TDD. The easier assignment in the second round obviously helped those with weak technical skills, but was perceived as less beneficial by the experienced students. So, the context and learning objectives must be carefully taken into account in teaching TDD.

Second, the students experiences raised up the question about the TDD as a design method. The design aspect came up in the easier assignment. In the more difficult assignment the students would have clearly needed design **before** starting TDD.

Finally, the second experiment confirmed our observation that the students are sensitive to new information. They seemed to believe in the claimed benefits of TDD despite the difficulties they perceived. We should pay attention to what and how we teach because the students may really believe it. This should also be taken into account when drawing conclusion from the student experiments.

Table 1: Experienced difficulties with TDD

Main categories	Explanation	Example quotes
TDD approach	adapting to the test first approach discipline in short cycles	”The most difficult was to rigorously write tests before the functionality.” ”I couldn’t naturally divide the work into small pieces. Often I wrote one major test or several tests at once and then used an hour in coding the functionality.” It is so easy to slip from TDD.”
Designing of tests	designing appropriate tests generally designing test in small steps	”Often I didn’t find any reasonable test to continue with.” ”It was not always easy to come up with good and consistent test cases, particularly without any specification of the problem” ”To come up with a proper test was almost always more difficult than writing the corresponding production code”
Technical difficulties	general programming skills Java experience JUnit or Eclipse	”Everything was difficult. I had never used Java before and my last programming experience was 10 years ago.” ”The first problem was learning JUnit environment.”

7. REFERENCES

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1st edition, October 1999.

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

[3] S. H. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proc. Int”l Conf. Education and Information Systems: Technologies and Applications (EISTA 03)*, 2003.

[4] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions on*, 31(3):226–237, March 2005.

[5] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, Sept. 2005.

[6] R. Kaufmann and D. Janzen. Implications of test-driven development: a pilot study. In *OOPSLA ’03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299, New York, NY, USA, 2003. ACM.

[7] K. Keefe, J. Sheard, and M. Dick. Adopting XP practices for teaching object oriented programming. In *ACE ’06: Proceedings of the 8th Austalian conference on Computing education*, pages 91–100, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[8] S. Kollanus and V. Isomöttönen. Test-driven development in education: experiences with critical viewpoints. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 124–127, New York, NY, USA, 2008. ACM.

[9] W. Marrero and A. Settle. Testing first: emphasizing testing in early programming courses. In *ITiCSE ’05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 4–8, New York, NY, USA, 2005. ACM.

[10] R. Martin. Bowling game kata, 2005.

[11] M. Muller and O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings -*, 149(5):131–136, Oct 2002.

[12] M. M. Müller and A. Höfer. The effect of experience on the test-driven development process. *Empirical Softw. Engg.*, 12(6):593–615, 2007.

[13] M. M. Müller and W. F. Tichy. Case study: Extreme programming in a university environment. In *23rd International Conference on Software Engineering*, pages 537–544. IEEE Computer Society, 2001.

[14] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Softw. Engg.*, 13(3):289–302, 2008.

[15] D. H. Steinberg and D. W. Palmer. *Extreme Software Engineering A Hands-On Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[16] A. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Newbury Park, California, 1990.

APPENDIX

A. QUESTIONNAIRE

The questionnaire, which we used to collect the experiences, included the following questions.

1. How much working experience do you have on ICT field? (This was not asked in the first experiment round.)
___ years and ___ months

2. Had you tried TDD before the course?
1 yes
2 no

If yes, where have you used it?
1 in my work
2 in another course
3 other, describe ...

3. How strictly did you follow TDD in the assignment?
1 very strictly
2 I/we slipped few times
3 I/we slipped occasionally
4 I/we slipped often
5 I/we did the code first and the tests after it

4. How did you organize your work (if you worked with a pair)?
1 we worked together using the same computer
2 we shared the work and finally put the parts together
3 partially both 1. and 2.
4 only one of us did the work

5. How hard was it to use TDD with scale from 1 to 5 (1 = not at all, 5 = very hard)?

6. Would you like to use TDD after this experience? Reason your answers.

7. What was most difficult in the assignment?

Why Using Robots to Teach Computer Science can be Successful Theoretical Reflection to Andragogy and Minimalism

Marja-Ilona Koski
Department of Computer Science
University of Helsinki
Finland

Jaakko Kurhila
Department of Computer Science
University of Helsinki
Finland

Tomi A. Pasanen
Gamics Laboratory
Department of Computer Science
University of Helsinki
Finland

ABSTRACT

To help students understand subjects such as theoretical aspects of computation, algorithmic reasoning and intelligence of machines, a number of publications report experiments to teach these topics with the help of Lego Mindstorms robots. In the publications, the researchers report how they have created various ways to approach the issues either in Computer Science or in Artificial Intelligence. The reported results of the experiments are based on the learning outcomes, the feedback from the students, and the perceived informal observations (i.e. "feelings") of the instructors.

But can anyone else benefit from the reportedly positive outcomes of the experiments? To give an answer to that question, this paper analyses the reported results through two support theories. The two theories chosen for this, andragogy and minimalism, are concerned with adult learning and how teaching adults should be approached. When reflecting the results of the four teaching experiments to the suggestions drawn from the theories, a more comprehensive answer to why the experiments have been successful can be given.

The four teaching experiments analysed here were in many ways similar to each other. A connection to the chosen support theories was straightforward to make. Besides describing the artefacts of teaching with the robots, a deeper discussion on this teaching approach is provided. For an instructor, all these observations offer more concrete evidence about beneficial factors of teaching with robots.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education – collaborative learning;
K.3.2 [**Computers and Education**]: Computer and Information Science Education – c*omputer science education, self-assessment*

Keywords

Robots, teaching experiment, adult learning theories, adult education, evaluation.

1. INTRODUCTION

It has been noted that hands-on experience is largely missing from the Computer Science classes while such sessions are common in laboratory sciences (Stein 1998). Some university instructors have chosen a different method to teach basic concepts from their field of specialization. They have decided to give Computer Science or Artificial Intelligence (AI) courses with the help of robots. These researchers have stated that this method helps to make the learning more captivating and interesting (Kumar and Meeden 1998, Imberman 2004 and Kumar 2004). However, none of these teaching experiments reflect their results on anything else than the experience itself and the responses given by the students. When reading these publications, an instructor willing to use such a method in his/her classroom might confront questions whether it is possible to produce the same positive learning outcomes again and does it really engage students to the task in the same way as described in the publications.

The aim of this paper is to clarify *why* the methods used in the teaching experiments can turn out to be effective by reflecting the earlier experiments to theoretical frameworks in two well-known learning theories. The researchers of the chosen teaching experiments described positive and/or negative outcomes, but they did not adequately treat the question whether there is a transfer of success if someone else decides to use the Lego Mindstorms robots in a similar fashion.

Four well-reported teaching experiments with Lego Mindstorms robots were chosen for our reflection (descriptions of teaching experiments in Subsections 3.1-3.4). In addition to the chosen experiments, there are of course several other publications that report the development in the field of the teaching with robots by designing new exercises for the class or innovative ways to use Mindstorms robots (Flowers and Gossett 2002; Imberman and Klibaner 2005; Imberman 2005; Jipping et al. 2007; Klassner and Continanza 2007). There are also other evaluation reports on the use of robots, but the focus of these is more on the robots themselves than on the learning process (Challinger 2005; Gross and Power 2005).

2. SUPPORT THEORIES

The theoretical frameworks used in our paper are andragogy and minimalism. The theories are well-suited for university-level education, as the learners are adult and the efficiency of education tends to be of high priority in formal educational settings. In the following two subsections these theories are explained and described on such a level that further evaluations of the chosen teaching experiments can be understood.

2.1 Andragogy

The theory of andragogy is based on a set of assumptions that describe how adults learn. This idea originates from the fact that adults learn differently than children, and the pedagogical methods used to teach children will not work among adults. The first of assumptions of the theory is *self-concept of the learner*. It describes how it is the job of an adult educator to move the self-concept of the learner from being a dependent personality towards the self-directed learner (Knowles 1980). The theory of andragogy directs the instructor to recognize the duty to encourage the adult learner to move away from his/her old learning habits, and to become a self-directed, independent learner who takes responsibility of his/her own learning activities.

The second assumption is *prior experience of the learner*. Here, the learner's experience of life is taken into account when new concepts are taught. Adults have a reservoir of experience which is a rich resource in the learning process to themselves and for others (Knowles 1980). Knowles (1980) states that people attach more meaning to the studied matter if they gain it from the experience than if they acquire it passively. Thus, an ideal situation for an adult to learn is with the laboratory experiments, discussions, problem-solving cases, simulation exercises and field experiences (Knowles 1980).

Thirdly comes *readiness to learn*. The idea of the third assumption is to evoke the learner's need to know the matter being taught. People become ready to learn something when they realize that they need to know it in order to perform better with real-life tasks (Knowles 1980).

The fourth assumption is *orientation to learning*. In adult teaching, it is important to acknowledge the fact that adults want to apply the knowledge and skills they learned today into living more effectively tomorrow (Knowles 1980). Adults need to find out what kind of effect the newly learned skill will have in their everyday life. Because of this, adults tend to learn in a problem-centered or performance-centered way of thinking (Knowles 1980).

Learner's need to know is the fifth assumption. Adults want to know the reason why something is important to learn, and how they can benefit from it (Knowles, Holton, and Swanson 1998). The adult learner needs to value the lessons, and his/her expectations should be filled in the class room by including an explanation of the importance of the matter.

The sixth assumption, and the last one, is *motivation to learn*. It is important for an adult educator to realize that potential motivators of the adult learning process are internal, and they come from the learner's own experience of him/herself (Knowles, Holton, and Swanson 1998). This does not exclude the fact that adults also respond to external motivators. Such factors as self-esteem and quality of life are important in giving adults a reason to learn (Knowles, Holton, and Swanson 1998). Expressing the learner's own opinion of the prioritization of the topics covered in class can give a learner the needed boost of motivation to learn.

2.2 Minimalism

The theory of minimalism assumes that when people are engaged in a task they will start to reason creatively and improvise (Carroll 1998). To support this, only the metadata of the matter should be provided to the learner, so that the learner can make the assumptions and reason on his/her own. The second thought derives from the idea of creative reasoning and improvising. When people are creating something or going with their instincts, errors tend to occur. This kind of action path is supported by the theory of minimalism. If and when errors occur, they are recognized and diagnosed with the help of the instructor and the use of a text book or a manual (Carroll 1998).

The first of the principles advises to *choose an action-oriented approach*. Opposite to the typical manuals where the first task is assigned on say page, 15, a manual designed as minimalism suggests introduces the first task on page one or two (Dubinsky 1999).

The second principle is *anchoring the tool in the task domain*. If the documents are done according to the suggestions given by minimalism, the text is presented in a short and simple way, and it must be easy to understand (Dubinsky 1999). The chapters are designed for an average user without long introductions and technical descriptions.

Support of error recognition and recovery is third on the list of principles. The idea behind this principle is the assumption that beginners make mistakes. The intention is to make features into the documentation that help the learner to identify and recover from his/her mistakes (Dubinsky 1999). This way, learning is a process where the learner is directing his/her own learning. The whole process should be seen as discovery learning where the learner is active and highly motivated by the tasks (Carroll and van der Meij 1996).

Support reading to do, study and locate is the last of the principles. The goal in this is to keep every section of the text self-contained (Dubinsky 1999). With the independent parts, the learner is not confused by the cross-references to earlier or later chapters. The main idea, when designing manuals, should be to give a learner the possibility of sequential processing, but also to enable random access approaches as well (Carroll and van der Meij 1996).

3. EARLIER TEACHING EXPERIMENTS

The goal of the following four subsections is to give the reader an idea about the evaluated experiments. Before looking in more detail into the teaching experiments, a short description concerning the tools used is needed. All of the chosen teaching experiments used the same Lego Mindstorms robots (Lego 2007) to teach the desired notions of the Computer Science.

The Lego Mindstorms kit includes Lego bricks to build the robots, and one programmable Lego brick called RCX. The first three experiments used the RCX brick to control the robot. Contrary to the others, the fourth teaching experiment used the Handy Board (Handy Board 2003) as a central unit of the robot. The Handy Board is a microcontroller system for building small, mobile robots mainly for educational or hobbyist purposes.

In the case of RCX, all programs are downloaded to it from a computer through an infrared transmitter which is connected to the computer's USB port. The RCX brick has three outputs A, B and C for the motors and for the lamps, and three inputs 1, 2 and 3 for the sensors. This Lego Mindstorms kit includes a Windows-based visual programming environment, but in all of the experiments it was stated to be too limited in its expressive power for the tasks that needed to be accomplished. The Handy

Board works almost in the same way; it only has a few improvements compared to the Lego product.

3.1 Teaching a Computer Science course in the US Air Force Academy

The teaching experiment described in this section is based on the following three publications (Fagin 2000, Fagin 2003, and Fagin, Merkle, and Eggers 2001).

The goal of the Computer Science course is to provide the learners with a strong core competence for future Air Force officers. One of the desired skills is programming. The authors argue that the use of the Ada/Mindstorms and the robots offer a new and interesting way to teach basic computing and controlling concepts.

The course was about introducing basic computing ideas, such as sequential control flow, selection, iteration, input/output, arrays, graphics, procedures and file processing. Six of these concepts are introduced in the publications.

Sequential control flow was taught with an exercise where the students were given a robot with two wheels connected to outputs A and C and the task was to write a program that makes the robot go forward for two seconds, then play a song, and after that go forward for one second, and stop.

To teach the use of variables, the method used was to show how the robot changes its behaviour according to the quantity in question. To demonstrate the meaning of the term in action, an exercise was made where the amount of time that the robot travels is changed by a numeric calculation written in a program code,

The benefit of the use of constants was demonstrated to the students by a problem in which the robot needs to turn right 90 degrees. An accurate amount of time required for an accurate 90-degree turn is represented as a constant.

All the programs consist of procedures. When writing a program for the robot, it can be seen that the smaller tasks, such as the one presented previously, should be written as a procedure. One problem can be reduced into smaller problems and this way reaching the goal step-by-step is easier than solving the problem at once.

To approach selection and Boolean expressions, authors have chosen a task where the Mindstorms robots react to their environment. Robots can receive inputs though previously stated input ports 1, 2 and 3 and this way the behaviour of the robot can be controlled and influenced.

In a case of teaching arrays the students were asked to capture a sequence of numbers which were given as input through the presses of touch sensor and bumper. Once this sequence was captured, it was the robot’s job to “play it back”. In other words, it means that the robot needs to examine each number and execute the part of the program where there is a predefined action to that number.

3.2 Use of robots in an Artificial Intelligence course at Ramapo College

The teaching experiment described in this section is based on the following two publications (Kumar 2001 and 2004).

The course was a traditional AI course in the sense that the students were taught representation and reasoning, focus on search, logic and expert systems. The first time the course was held this way was in the fall semester 2000. The results about teaching the course and the learning experiences have been monitored for three years. During that time the basic concepts and ideas of the course have stayed the same, only the minor adjustments stated before were done.

The first task in the AI course was a project on blind searches; depth-first and breadth-first search of a tree. At the beginning of the programming task the students could assume that the tree is a binary tree with tree levels. However, in the later stage they had to generalize their implementation to deeper trees with an arbitrary branching factor.

The second task was about heuristic searches; hill-climbing and best-first search. In the project, the robots were expected to find their way out of a maze. In addition, while searching for the route out, they were expected to build a search tree of the maze. The robots interacted with their environment through touch and light sensors.

In the third task, the robot had to be able to determine the characters printed on a grid. To be able to complete this project, the students had to use the idea of forward and backward chaining in a rule-based expert system. The robot had to be able to go through a grid of pixels and use both the forward (data-driven) and the backward (goal-driven) chaining to determine the character in question.

3.3 An elective AI course in Villanova University’s Computer Science program

The teaching experiment described in this section is based on the following publication (Klassner 2002).

It should be noted that Cognitive Science minors and Computer Engineer students also participate in the course. These students have no programming experience or at the most, a one-semester course of introduction to Java. However, the Computer Science majors, who participate in this course, take it in their fourth year, and by then they have taken a course “Programming Languages”.

The first project of the course was a one-week project where the students experienced that with the simple stimulus-response rules and a limited model of the environment, the robot could achieve effective behaviours. The first task was to program the robot to move randomly ignoring any stimulus coming from outside. When this behaviour was accomplished, the robots were timed on a short obstacle course with narrow passages.

The second task was that the robots needed to monitor their environment. The feedback from the outside world to the robot came through an infrared, a light or a touch sensor. The infrared or light sensors were used to determine whether the robot was too close to the wall. The light or touch sensors were in use to detect if the robot has wedged into a corner. After either of these changes, the robots were timed on the same course as in the first part of the project. The result from the first task was now compared to the results of the second task and this way the students could observe the improvements.

In the second project, the goal was to show students how sensitive each of the sensors were to various stimulus. The goal was also to demonstrate how some sensors can interfere or simulate other sensors’ capabilities. The first part of this project required a team of students to work with all types of sensors (touch, light, infrared and rotation) to generate different kinds of inputs and this way to study the various responses that the sensors could generate. The second part of the project was to design in teams a simple robot that used only a touch, an infrared and/or a light sensor to duplicate the accuracy and sensitivity of the rotation sensors. With this, the purpose was to demonstrate the concept of the functional emulation.

The third project was a project of two weeks where the students acquainted themselves with an important issue in the navigation process. The goal was to help the students understand factors that could cause error in the robot's internal representation of where it thought it would be located in the world.

Projects four and five were about building a ball-playing robot to compete against the robots built by the other teams. In these projects the students combined the previously learned skills, but also encountered new problems and possible solutions. In the project four teams built only one robot that played the game against the other robots, but in project five, three robots per team entered the ball field.

The sixth and the last of the projects was a two-week project with the goal of showing the students that the knowledge representations that speed up the search-based problem solvers can produce such a solution presentation that cannot be easily translated into the control programs of the hardware. In the first part, the students had to solve an 8-Puzzle by developing a knowledge representation and a Lisp search program. The teams developed a set of four operations to conceptually move the robot. These four movements reduced the search compared to 32 operations that would be needed to move each of the numbered tiles. In this way the students could experience the reduced branch factor of the search tree, leading to a faster execution time for the game solver. In the second part, the students were asked to write a program that invoked functions in an ad hoc library, developed by the instructor, to send remote-control messages to the robot's arm mechanism and this way move pieces on the 8-Puzzle.

3.4 An elective AI course in College of Staten Island

The teaching experiment described in this section is based on the following two publications (Imberman 2003 and 2004).

The chosen method to control the robots on the course was the MIT Handy Board. The controller of the Handy Board contains 32K of battery-protected RAM, and it has four DC motor outputs, nine digital and eight analogue inputs. These inputs support diverse sets of sensors. Several compilers are available for the Handy Board, including the Interactive C. Using the Handy Board for the Lego-based robots enables creation of sophisticated behaviours. Because of this, the Interactive C was chosen as a programming language to the project.

The overall objective of the project was to design and build a robot that will use a neural network to successfully navigate a circular path. The project started with the instructions on how to build a gear box for the robot to work properly. Instructions were also provided to build the robot so that it would be suitable for the path-finding task. Even though building the robot is an important part of the whole project, the goal here as well as in the other teaching experiments was not to spend too much time on concrete construction of the robot.

The third part was to write a program with the Interactive C where the robot moves forward for half a minute, then turns right, and then again goes forward for half a minute and turns, this time to the left. The goal was to find a minimum motor power to make the robot move. For the training examples of the neural network, the turning power needed should be written down.

In the fourth task, the students first wrote a program that would display the readings from both the robot's photo sensors. Again the readings should be written down because they would be used later in the project. To continue the project, the students had to estimate the parameters needed for the left and right motor functions to control the real wheels.

After these four tasks, the students used the earlier modified Generation 5 code to program their robot with a neural network. Once the training was done, the generated neural network was tested on another robot with the same type of sensors. When the students had the weight values for the neural network equations, they incorporated them into the Interactive C neural network and tested their robot. It was important at this phase to make the robot move slowly enough that it would have enough time to take the readings from the road and act upon them.

4. ANALYSIS

4.1 From mistake to understanding

In the results from the chosen teaching experiments, researchers report a better learning outcome in certain topics which are usually considered to be difficult to the students. During the AI course, if the students were asked to make a conceptual difference between training a neural net and a finished product, a trained neural network, they usually had problems in answering the question (Imberman 2003). Due to the architecture of the robot, there was not enough memory to train a neural net on the robot. Students soon realized this, and they started to do the first part on a computer and then transfered the finished product to the robot (Imberman 2003).

Similar results were observed when the students were learning the concept of procedures. Students added a new procedure to the code, but forgot to call it in the later stage of the code (Fagin, Merkle, and Eggers 2001). As a result, the robot did not present its newly added behaviour. Because the robot visualizes the commands in the code the students could observe the incompleteness of the code immediately and that helped them locate the problem.

Learning, as it is described in these experiments, can look like learning by trial-and-error. However, it can also be seen as a learning process where the learner is directing his/her own learning. It was stated that students thought they used the procedure call correctly, and only after testing found out what was missing (Fagin, Merkle, and Eggers 2001). Researchers also stated that when the students looked through the sequential control flow of their program code, they immediately saw that they did not program the robot to do its new behaviour (Fagin, Merkle, and Eggers 2001). So when the robot's actions were not the wanted ones, students needed to reconsider the solution. According to the theory of minimalism, beginners make mistakes, and the use material should support error recognition and recovery from mistakes (Carroll and van der Meij 1996). When the action path is as it is described in the experiment, the robot supported the recognition of the error by showing the missing part in the program code with its behaviour.

4.2 Designing a course

Traditional courses create a more comfortable learning environment because the instructor has years of experience in what to teach and how to teach it. How well the instructor handles the studied topic and the study material is reflected by the students' experience of the course (Fagin and Merkle 2002). However, the little amount of experience can also be turned into the strength of the course, and as a possibility for the learner to take charge of his/her own learning activities.

Students often feel that education is something done to them instead of experiencing it as something that they are actively doing for themselves (Beer, Chiel, and Drushel 1999). With the change in the attitudes of the students, the encountered situation of uncertainty could be seen as an instructor's way of supporting the students to become independent and self-directed learners. The theory of andragogy states that the problem with the adult learner is a learning model from previous schooling (Knowles 1980). A more familiar approach to students is to get the answer of what to do than to figure it out by themselves. Also, based on the same theory, the adult learner has a need for autonomy (Knowles 1980). Therefore, by providing guidance to the learner, the instructor can be more beneficial in the learning process than by being a person telling students exactly what to do.

The theory of minimalism also suggests that the manuals used for studying would not be totally complete (Carroll and van der Meij 1996). This does not mean that the students are left without any guidance or help, but to encourage them to use their abilities and knowledge to "fill in the gaps". The material designed to help the students solve their problems should give enough support but also leave space for their own interpretations and ideas (Carroll and van der Meij 1996).

4.3 Workload of the course

With the Mindstorms robots the workload of the course is bigger than course credits may predict (Klassner 2002). Because many universities are not willing to raise the number of credits gained from the course, instructors had to make a decision that the course will have an open lab work (Kumar 2004) or allow the students to take material out of the lab to work on it at home (Imberman 2004). Contrary to the author's beliefs, the students did not consider this a drawback of the course. The students reported that they spent a vast amount of time on constructing the robot and testing their code, but by the end of the course, it all appeared to them as a good investment (Kumar 2001).

The reason why students considered the workload of the course rewarding could be that the students were ready to learn the subject that was taught. The theory of andragogy describes how to evoke the learner's readiness to learn (Knowles 1980). To make the learner realize the importance of a certain knowledge or skill, an instructor can design experiences of the situations, where the learner needs that knowledge or skill.

The same theory states that adults become ready to learn something when they realize that they need the knowledge to cope better in real life (Knowles 1980). Working with robots can be also seen as an answer to why he/she needs to learn it. It is important for an adult learner to have a reason why he/she needs to know the subject (Knowles 1980). With the robots the studied matter made more sense because the abstract theory or algorithm was presented in a way that students could relate to.

With robots it is easier to create an image of a situation in real life than with a program that is only showing something on a computer screen. Working with robots, students face the non-idealistic situations where the real-world problems occur (Beer, Chiel, and Drushel 1999). However, this happens in a safe environment. Furthermore, the use of robots associates computers to toys and this way reduces any possible fear of trying out and exploring (Lawhead et al. 2002). This way the learning situations with the robots are seen as an opportunity, and the effort put into them is worth it.

4.4 Teamwork

More complicated assignments invited the students to start working in groups. This was due to the large workload of the projects from the beginning. Forming groups showed that they became more competent in estimating how much time completing a project actually takes. This happened in a sense that students started to set more realistic goals for themselves compared to the beginning of the course (Kumar 2001).

Moreover, the theory of andragogy talks about using the experience of the learner as part of the teaching (Knowles 1980). The experience should be seen as a starting point to the learning process (Knowles 1980). When the students worked in teams or discussed their solutions, they used someone's experience of something. In that sense, the previously mentioned adaptation can also be seen as a result of using the expertise knowledge of what different fields of studies or different specialization directions provide. It can lead to a better adaptation to the subjects in later courses or, as students in one publication have reported, the course problems had a positive influence on their learning (Imberman 2004).

In addition, there is an interesting possibility for the students to learn to express their ideas, but also to give and receive criticism. Students learn a valuable lesson if they see that the variety in perspectives can be helpful for solving a hard problem (Beer, Chiel, and Drushel 1999). In one of the publications, the students reported that in the beginning they had doubts about the usefulness of the course, but by the end they admitted that the course offered useful skills for the future (Imberman 2004). This course offered an opportunity to naturally work in groups, making it possible to practice both Computer Science and social skills for future needs.

4.5 Building the robots from the model

One of the problems in using the robots to teach Computer Science concepts is finding the balance between how much time can be consumed on building the robot and how much on programming. Building the robot can be fascinating and inspiring, but it can also be time consuming and frustrating. Some of the authors have resolved this problem by giving instructions on how to build the robot, and simply minor moderations are left to the students (Imberman 2004, Klassner 2002, and Kumar 2004).

But can the robot still serve the same purpose as a factor of inspiration in the learning process if the model to build the robot is given to students? It is stated that it was difficult to make the robot behave reliably (Kumar 2004), projects were more difficult than expected because the sensors did not work reliably enough

(Klassner 2002), and adjustments needed to be done in the testing phase to both, the robot and the testing surface (Imberman 2003). So reflecting this to the problem of whether giving the instructions to build the robot or not is justified.

Experience states that students still became enthusiastic about building the robots even if it was just the customizing and fine tuning (Imberman 2003). The reaction of the Computer Science students also supports this when they have written on the feedback form that instructors should spend more time on planning how to build the robots, so that the time spent on designing could be reduced (Klassner 2002).

4.6 Learning more than what was taught

When analysing the exit surveys of the courses, the researcher noticed that the students learned concepts outside the curriculum. Klassner (2002) reports that the students were more confident about their skills to do multithreading tasks after taking the course with the robots than before when the course was taught in a more traditional way. Besides learning the desired notions, the students were able to obtain a skill to evaluate that their knowledge is sufficient.

According to the theory of andragogy, motivation for learning comes from the learner's own experience of him/herself (Knowles, Holton, and Swanson 1998). Because of this, it is relevant for an instructor to acknowledge the learner's need to have trust in his/her own abilities. With adult learners especially, these internal motivators are the most important (Knowles, Holton, and Swanson 1998). In this case, studying with multitasking programs for the robots became an accelerator for moving on to more complex domains. Because of the nature of the robot problems and solutions, the need to try something more challenging comes naturally.

4.7 Orientation to learning

The students evaluated that working with the robots helped them in understanding the complexity issues of the algorithms (Kumar 2004). With the experience of testing and seeing what the result is, the students could be able to see right away what the behaviour would look like. The robots create a performance-centered atmosphere for the learning, which is an ideal environment for adults to learn according to the theory of andragogy (Knowles 1980).

The robot can be seen as something interesting to apply the newly learned skill to. The robot offers an incentive to learning because students want to see their invention succeed (Kumar and Meeden 1998). When developing the right solution, students experience many different variations of a possible solution. Because students have a need to find the best possible solution to a problem they have encountered, the explanation for the search comes from their own need. This motivates students to learn about the less glamorous theoretical aspects of Computer Science (Kumar and Meeden 1998). With this method students are introduced to new aspects of theories behind the solutions, and they encounter aspects that may not be visible in the normal search of a solution. A researcher writes in his publication that the students stated that after the course, they have learned how to apply an algorithm to a certain problem (Kumar 2004). The process where the understanding of a problem becomes clearer little by little could be seen as a reason why the students were able to choose the right algorithm to a problem.

5. DISCUSSION

Working with the robots offers a chance to implement the code as a real-world construct. It offers a unique possibility to test the design in action right away with a minimal effort. The programmers with ten years of experience have complained that young programmers depend too much on the technology in order to complete the tasks given to them (Wolz 2001). When error is seen in action, it introduces a possibility to the students to test every modification of the code on the robot. In one of the teaching experiments, it was stated that students thought their solution was correct before testing it on the robot (Fagin, Merkle, and Eggers 2001), but can it be proven that it did not happen in all the other cases? Because if it does, it proves that designing before doing is still a skill that was learned in the old days when a batch submitted to be compiled required two days of waiting (Wolz 2001).

As much as teaching with robots has been praised, it has also been criticized. Learning to program through trial-and-error can easily be compared to learning with robots. However, it has been shown that the students tend to consider the decisions they made when writing the code, and after that they transfer fully ready solutions to the robot (Imberman 2003). Also Kumar (2004) reported that the students have shown better understanding of the complexity issues of the algorithms, and the results of the tests have revealed better knowledge of how to apply an algorithm to a problem. So if the students have the ability to decide and design the correct solution to a problem and according to that start executing their answer into a program code, it gives enough proof that code designing and management can be taught as well in the 21st century.

As much as the programming languages have developed, the platforms and programs have improved. Being able to perceive the outcome of the code is an important skill to master, but testing a program is still different from mindless re-testing. Nowadays, there are different techniques to do the coding and testing, and because of the nature of the robots, the test results give reasonable feedback and with this they direct correction in the right direction. Re-testing and negative outcomes can provide important lessons (Wolz 2001). Therefore, the whole concept of teaching and learning with robots should be seen differently. The traditional approach to programming gives students few opportunities to observe the behaviour of their code in any other context than in the debugging phase (Stein 1998). In this sense robots should not be used only to give hands-on experience, but to create an atmosphere that resembles something from real life. For future needs, it is important for the students to see how the environment around the robot affects the design. So maybe re-testing should not be compared to the designing of the code, but it should be seen as testing what effect the outside world has on the design.

With robots, the designing and implementing invites students to think of more options for how to plan the code to solve the problem, and with that students experience more aspects of the concept. When teaching is done this way, it invites students to consider not only how to build the program, but to think about what the behaviour will be and modify that behaviour (Stein 1998). Not only is programming as a skill hard to achieve, but the science of programming also includes a lot of details which are not easy to explain, nor is it easy to give a reason to the students why they need to learn them (Lawhead et al. 2002).

When teaching with robots, the programming is not a separate phase of the project, but it is attached to many parts of the project, such as designing and testing. The validation of the programming comes naturally with robots because the students are eager to see the robots work in action (Lawhead et al. 2002). The importance of connecting all these parts and making them work together is acutely present with robots. The construction of a physical entity joined with the code designed by the students themselves gives a unique opportunity to directly confront the central issues of Computer Science (Kumar and Meeden 1998). After students have designed a working robot, they have experienced some of the convergence of Computer Science, and thus can better perceive the interplay between various concepts. This is crucial because understanding the interactions between the program and its behaviour is critical in modern applications (Stein 1998).

Because of the small amount of research done on the use of Mindstorms robots to teach bigger concepts in the field of Computer Science, it was significant that one out of four reported a negative outcome. Some insight about this unsuccessful experiment by the US Air Force Academy has already been given in the analysis part of this paper. However, there are other observations as well that might explain the reasons behind the failure in the experiment.

For the limited amount of money to be spent on the robots, the researchers had to make a decision to use the robots only inside the classroom (Fagin and Merkle 2002). Even with the effort of giving as many lab sessions as possible, the simulation and testing phase was too short to make the use of the robots worth while. Researchers admit that in their experiment they were not able to give enough resources to one of the most important parts of the development of the robots (Fagin and Merkle 2002). The students also saw the unlimited time reserved for the projects as a big disadvantage. In a more traditional class, the way subjects are presented is a result of many years of teaching and examining student feedback. The reason why students in the class with robots showed worse results than those in the class without them (Fagin and Merkle 2002), could lie in this limited amount of resources.

Instructors of the Air Force Academy Computer Science course report that their students are not representative of the whole population of students, and hope that other researchers in different environments attempt a similar experiment (Fagin and Merkle 2002 and Fagin and Merkle 2004). Unlike the students in the other experiments, the students of the Computer Science course in the Air Force Academy had to design their code and built the robot within lab hours (Fagin and Merkle 2002 and Fagin and Merkle 2003); other researchers favoured and recommended to their students to use time more flexibly in their experiments. Even though the students in the other teaching experiments reported large time consumption on working on the robots at home, in the end it was considered to be rewarding, and a positive factor in their learning from both the students' and the instructors' point of views (Imberman 2003 and 2004, Klassner 2002, and Kumar 2001 and 2004). The researchers who received the negative result, acknowledge the fact that their choice to limit the time used on testing and debugging the robots is partly the cause why results were not what they expected (Fagin and Merkle 2002).

To defeat the ongoing competition inside of the university of which course gets enough students to enrol, robots can be one solution. Robots fascinate the typical student, and this interest should be used to invite students into the Computer Science curriculum (Kumar and Meeden 1998). Imberman (2004) reports that after starting to use robots in the AI course, the enrolment rate is better than before. Also Kumar (2001) reports that in the end survey when students were asked if they would recommend the course to their friends, over 90% answered yes. Besides this, those instructors who use robots in their class argue that they bring a fun factor to the class (Imberman 2004 and Kumar 2001). Even if university studies are not meant to be fun and entertaining, the experience of enjoying the class and having done exercises without feeling frustrated, should have a positive influence on the students' attitude towards studying.

The nature of the learning process is different when studying with robots than in more traditional ways. It could be considered as one option to create some variation in the Computer Science curriculum. We can still rethink the fundamental notions of computation in a way to bring teaching much closer to today's practice (Stein 1998).

This paper does not give an answer to the question what the best way to teach or approach an adult learner is. It only focuses on giving an explanation to why diverse methods could be taken into consideration when designing a course within the Computer Science curriculum.

6. CONCLUSION

These notions from the teaching experiments reflected on the support theories and diverse remarks give instructors a reason why to consider using robots in university-level education. The support theories give context-free responses to the observations reported in the publications. The positive or negative outcome can now be mirrored to the known behaviour or preference of an adult learner. With this an instructor can be confident that the outcome can be reproduced in his/her classroom. As a final aid for instructors we have collected and organized support theories and case studies in table form, see Table 1, for giving a summary of our research results.

7. REFERENCES

Beer Randall, Chiel Hillel and Drushel Richard (1999): Using Autonomous Robotics to Teach Science and Engineering, Communications of the ACM **42**(6):85-92.

Carroll John (1998): *Minimalism Beyond the Nurnberg Funnel.* 1-18. John M. Carroll. MIT Press.

Carroll John and van der Meij Hans (1996): Ten Misconceptions about Minimalism, *IEEE Transactions on Professional Communication* **39**(2):72-86.

Challinger Judith (2005): Efficient Use of Robots in the Undergraduate Curriculum, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 436-440.

Dubinsky James (1999): Fifteen Ways of Looking at Minimalism, *Journal of Computer Documentation* **23**(2):34-

47.

Fagin Barry (2003): Ada/Mindstorms 3.0: A Computational Environment for Introductory Robotics and Programming, *IEEE Robotics & Automation Magazine* **10**(2):19-24.

Fagin Barry (2000): Using Ada-Based Robotics to Teach Computer Science, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 5th annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, Helsinki, Finland, 32:148-151.

Fagin Barry and Merkle Laurence (2003): Measuring the Effectiveness of Robots in Teaching Computer Science, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, USA, 307-311.

Fagin Barry and Merkle Laurence (2002): Quantitative Analysis of the Effects of Robots on Introductory Computer Science Education, JERIC, *ACM Journal of Educational Resources in Computing* **2**(4):1-18.

Fagin Barry, Merkle Laurence and Eggers Thomas (2001): Teaching Computer Science with Robotics Using Ada/Mindstorms 2.0, ACM SIGAda Ada Letters **21**(4):73-78.

Flowers Thomas and Gossett Karl (2002): Teaching problem solving, computing, and information technology with robots, Journal of Computing Sciences in Colleges **17**(6):45-55.

Gross Paul and Power Kris (2005): Evaluating Assessments of Novice Programming Environment, ICER, The First International Computing Education Research Workshop, *Proc. of the 2005 International Workshop on Computing Education Research*, Seattle, WA, USA, 99-110.

Handy Board (2003): The Handy Board microcontroller system, http://www.handyboard.com Accessed 13 Aug 2008.

Imberman Susan (2005): Three Fun Assignments for an Artificial Intelligence Class, *Journal of Computing Sciences in Colleges* **21**(2):113-118.

Imberman Susan (2004): An Intelligent Agent Approach for Teaching Neural Networks Using LEGO Handy Board Robots, JERIC, *ACM Journal of Educational Resources in Computing* **4**(3):1-12.

Imberman Susan (2003): Teaching Neural Networks Using LEGO Handy Board Robots in an Artificial Intelligence Course, SIGCSE, Special Interest Group on Computer Science Education, ACM SIGCSE Bulletin **35**(1):312-316.

Imberman Susan and Klibaner Roberta (2005): A Robotics Lab for CS1, *Journal of Computing Sciences in Colleges* **21**(2):131-137.

Jipping Michael, Calka Cameron, O'Neill Brian and Padilla Christopher (2007): Teaching Students Java Bytecode Using Lego Mindstorms Robots, , SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 38th SIGCSE Technical Symposium on Computer Science Education*, Covington, Kentucky, USA, 170-174.

Klassner Frank (2002): A case study of LEGO Mindstorms'™ suitability for Artificial Intelligence and robotics courses at the college level, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 33rd SIGCSE Technical Symposium on Computer Science Education*, Covington, Kentucky, USA, 8-12.

Klassner Frank and Continanza Christopher (2007): Mindstorms without Robotics: An Alternative to Simulations in Systems Courses, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 38th SIGCSE Technical Symposium on Computer Science Education*, Covington, Kentucky, USA, 175-179.

Knowles Malcolm (1980): *The Modern Practice of Adult Education, from Pedagogy to Andragogy*. 40-59. Englewood Cliffs, Prentice Hall, Cambridge.

Knowles Malcolm, Holton Elwood and Swanson Richard (1998): *The Adult Learner: The Definitive Classics in Adult Education and Human Resource Development*. 2-5. Gulf Publishing, Houston, Texas.

Kumar Amruth (2004): Three Years of Using Robots in the Artificial Intelligence Course - Lessons Learned, JERIC, *ACM Journal of Educational Resources in Computing* **4**(3):1-15.

Kumar Amruth (2001): Using Robots in an Undergraduate Artificial Intelligence Course: An Experience report, 31st Annual ASEE/IEEE Frontiers in Education Conference, Reno, Nevada, USA, **2**:10-14.

Kumar Deepak and Meeden Lisa (1998): A Robot Laboratory for Teaching Artificial Intelligence, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 29th SIGCSE technical symposium on Computer Science Education*, Atlanta, Georgia, USA, 341-344.

Lawhead Pamela, Bland Constance, Barnes David, Duncan Michaele, Goldweber Michael, Hollingsworth Ralph, Schep Madeleine (2002): A Road Map for Teaching Introductory Programming Using LEGO Mindstorms Robots. The Annual Joint Conference Integrating Technology into Computer Science Education, Working group report from ITiCSE, Innovation and Technology in Computer Science Education, Aarhus, Denmark, 191-201.

Lego (2008): Lego Education – Mindstorms, the Lego Group, http://www.lego.com/eng/education/mindstorms/default.asp Accessed 13 Aug 2008.

Stein Lynn (1998): What We Swept Under the Rug: Radically Rethinking CS1, *Journal of Computer Science Education* **8**(2):118-129.

Wolz Ursula (2001): Teaching Design and Project Management with Lego RCX Robots, SIGCSE, Special Interest Group on Computer Science Education, *Proc. of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, USA, 95-99.

Table 1. Summary of analysis of case studies

		Case studies			
Support theory	Assumption/Principle	US Air Force Academy	Ramapo College	Villanova University	Staten Island College
Andragogy	Self-concept of the learner	In the class with robots, students were not encouraged enough to move away from their old learning methods. Students felt that their learning is related to the instructor's level of knowledge about the matter taught. The course failed to help the students obtain a more independent and self-directed way of learning.	Not mentioned in research reports.	The variation in the learning process helped students obtain more than what was expected from them in the course curriculum.	Instructors were able to encourage students so that they learned more than what was expected in the course curriculum.
	Prior experience of the learner	Not mentioned in research reports.	Learning more and realizing how much resources projects need students started to set more realistic goals for themselves than in the beginning of the course.	Course problems have positive influence on students' learning.	Course problems have a positive influence on their learning.
	Readiness to learn	Even though the robot projects were hard and time consuming, students pointed out that an advantage in working with them is that they make one want to learn and that they give an opportunity to learn something totally new.	According to the end survey: students spent more time on robot projects than on the traditional projects, but they also enjoyed them more.	Not mentioned in research reports.	Students felt that the course with robots offered useful skills for the future.
	Orientation to learning	Robot projects were described as mentally challenging. They are a great application to real life computing and students learned logical problem-solving skills.	Students evaluated that working with robots helped them understand complexity issues of algorithms. Students learned how to apply an algorithm to a problem.	Not mentioned in research reports.	Machine learning is a hard concept for the students to understand. With the traditional way of teaching, students may lose their focus on what is the real problem. With robots it was easier to demonstrate and this way point out the significance.
	Learner's need to know	Projects have been described as fun and magical, but it was unclear for the students why they needed to learn the matters. Students felt that projects were irrelevant for Computer Science majors and not practical unless one is going into that career.	According to the end survey: Learning and working with robots appeared to students as a good investment.	Students valued the matter taught and therefore the course had a positive influence on students' appreciation of the issues behind the design of agents.	The fun factor has introduced the course to students who normal would not take AI. Students have returned afterwards and express that the topics covered were relevant to their work experience.
	Motivation to learn	The projects failed to connect into the students' inner motivators. Students experienced that either one understands what the problem is or then one does not, but there is no in between. Even with instructors' efforts to provide as much help as possible, students felt isolated.	With the robot projects, instructors were able to get the students to use their imagination and therefore the learning method became more effective.	Because building actual robots and programming them as a constructive activity can be viewed as inherently motivating, especially because of the rapid feedback of success and failure, students became more confident about their skills and learned to evaluate their knowledge.	Working with robots offers practice that is better related to real-world problems. Therefore students reflected their learned skills as something that is useful in future studies and working life.
Minimalism	Choose an action-oriented approach	Exercises on the course were designed in a way that students used robots built from a model, to be able to start working faster on asked problems.	Exercises in the course were designed in such a way that students used robots built from a model, to be able to start working faster on required problems.	Exercises in the course were designed in such a way that students used robots built from a model, to be able to start working faster on required problems.	Exercises in the course were designed in such a way that students used robots built from a model, to be able to start working faster on required problems.
	Anchoring the tool in the task domain	Students were expecting more answers from the instructor and not supported enough to solve the problems by themselves or find answers of their own.	Robots were used to connect the matter better into something that presents the concepts in a way that students can relate to.	Robots were used to connect the matter better into something that presents the concepts in a way that students can relate to.	Robots were used to connect the matter better into something that presents the concepts in a way that students can relate to.
	Support of error recognition and recovery	Students looked through the program code, and they immediately saw that they did not program the robot to do its new behaviour. The robot visualizes the commands, so that the students could observe the incompleteness of their code immediately and locate the problem.	Robots visualize the previously hidden process of code execution. Due to the robots' instant feedback, error recognition is more straightforward and therefore recovery is improved.	Robots visualize the previously hidden process of code execution. Due to the robots' instant feedback, error recognition is more straightforward and therefore recovery is improved.	The architecture of the robots made the conceptual difference between different notions more visual than the traditional method used before.
	Support reading to do, study and locate	Dave Baum's book was used to help the students solve problems that occurred during the learning process.	Dave Baum's book was used to help the students solve problems that occurred during the learning process.	One part of the book of Russell and Norvig, was used to clarify the topics.	One part of the book of Russell and Norvig, was used to clarify the topics.

A Global Software Project: Developing a Tablet PC Capture Platform for Explanograms

Tony Clear
Jacqueline Whalley
Jonathan Hill
Yong Liu
Auckland University of Technology
Private Bag 92006
Auckland 1020
New Zealand
+64 9 921 9999
Tony.clear@aut.ac.nz
jacqueline.whalley@aut.ac.nz
fnx1465@aut.ac.nz
xzp1481@aut.ac.nz

Arnold Pears
Uppsala University
Box 325
75105 Uppsala
Sweden
+46 18 471 0000
Arnold.pears@it.uu.se

Beryl Plimmer
University of Auckland
P.O. Box 123
Auckland 1020
New Zealand
+64 2 9514 2000
Beryl@cs.auckland.ac.nz

ABSTRACT

Explanograms provide "a sketch or diagram that students can play" [10]. They are a directly recorded multi-media resource that can be viewed dynamically. Often they are used in teaching situations to provide animated explanations of concepts or processes. *Explanograms* were initially based upon proprietary paper and digital pen technology. The project outlined here augments that design by using a tablet PC as a mobile, general purpose capture platform which will interoperate with the existing server based system developed in Sweden. The design of this platform is intended to achieve both learning and research outcomes, in a research linked learning model for global software development. The project has completed an initial development phase during which a prototype has been built, and a consolidation, extension and evaluation phase is now underway. The origins and goals of the research, the methodology adopted, the design of the application and the challenges that the New Zealand based team have faced are presented.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education - *Computer science education*

General Terms

Human Factors

Keywords

CS Ed Research, Tablet PC, Digital Ink, Explanograms

1. INTRODUCTION

This paper presents progress to date on a global software development project which has been developing a general purpose platform for generating *explanograms* "a sketch or diagram that students can play" [10]. The paper addresses both product and process dimensions of the project. Although explanograms were originally conceived as "an animated presentation generated by writing on a sheet of paper" [11], this project aims to augment the present proprietary paper and digital pen technology available for explanograms, by incorporating a tablet PC as a mobile and general purpose capture platform. The purpose of an explanogram has been explained as follows:

> *"Explanograms originated as an approach to capturing multi-media versions of impromptu explanation; thus making them available to a wider audience. The underlying assumption is that difficult areas of the curriculum are often those that prompt students to present themselves during staff "office hours" and ask a question" [11].*

While the original goal for explanograms was to enable these 'snippets of wisdom' to be made available to wider student audiences for learning purposes, the project described here is also motivated by research interests. It is concerned with the potential value of explanograms in Computer Science education research in capturing student work in progress within natural settings, with a specific interest in supporting the work of the BRACElet project [18]. The adoption of the Tablet PC technology is seen as both a mechanism to support mobile learning and a further means of making the explanograms available to a wider audience of educators and researchers.

The project occurs in the context of a global research and development collaboration. The explanogram concept was developed at Uppsala University in Sweden, and this extension project has originated from Auckland University of Technology (AUT) in New Zealand. Supported by a grant from Microsoft

Research Asia, after an introductory Microsoft hosted workshop in Singapore, the initial proof of concept and experimental prototype development took place over summer 2007 with the assistance of a research assistant who developed an application to run explanograms on the Tablet PC. The present stage of the project is being carried out as a capstone Research & Development project within the Bachelor of Computer and Information Sciences at AUT. Thus the project provides an example of a collaborative and "research linked teaching and learning model" [4]. In this stage the team are refactoring the code, adding functionality, improving the user interface and the host connectivity. These initial phases could be likened to the "Research prototypes" stage of the technology maturity life-cycle proposed in [1] and portrayed below.

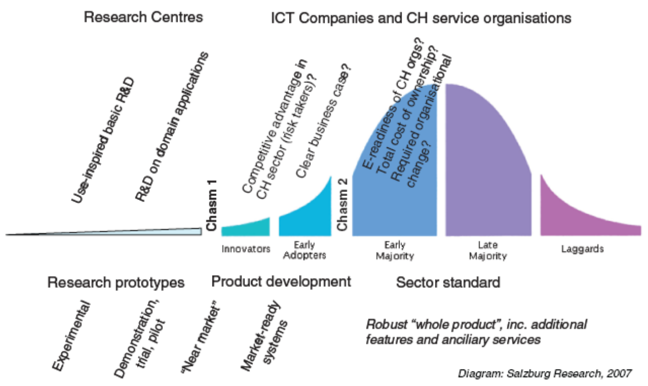

Research Centres
ICT Companies and CH service organisations
Use-inspired basic R&D
R&D on domain applications
Competitive advantage in CH sector (risk takers)?
Clear business case?
E-readiness of CH orgs? Total cost of ownership? Required organisational change?
Chasm 1
Chasm 2
Innovators
Early Adopters
Early Majority
Late Majority
Laggards
Research prototypes
Product development
Sector standard
Experimental
Demonstration, trial, pilot
"Near market"
Market-ready systems
Robust "whole product", inc. additional features and anciliary services
Diagram: Salzburg Research, 2007

Fig. 1. Phases and chasms in the technology maturity life-cycle model [ex. 1, p.12]

The above model of technology maturity "builds on the standard model of the diffusion of innovations [Rogers 1962, 1995] but includes phases that are often not considered" [1]. The early R&D based phases are included in the above model, which expands upon the 'research phase' as opposed to the traditional focus on the later commercialisation and adoption phases. The model also identifies a major chasm between conceiving and developing research prototypes, and their adoption by "by one or more innovative companies in search of a competitive edge" [1].

The relevance of this model of technology diffusion for the project reviewed here further relates to the learning involved for one of the authors who is studying a capstone project for his conjoint degree with both Software Development and E-Business majors. The E-Business aspect of the project involves researching a combination of Open Source Software [14] and "Software as a Service" [9] business models for traversing the first chasm above, and making the results of the research usable by a wider community of educators.

2. PROJECT GOALS

This section draws heavily upon the submission for funding made in response to the Microsoft Research Asia - 2007 Mobile Computing in Education Theme, and presents the broader scope of the planned research within which this project is one element. The research programme as proposed aimed to use Tablet-PC and Microsoft digital ink technologies to leverage three current Computer Science Education Research strands, across several institutions. The first of these, the BRACElet project [18] is a multi-institutional, multinational project investigating novice programmers; the second arising from the work of Arnold Pears at Uppsala University in Sweden involves the notion of an explanogram "a sketch or diagram that students can play" [10]; and the third involves international collaborative studies with Global Virtual Teams [4].

The proposal was not aimed at developing a discrete one-off application for the Tablet PC with potential for educational use. Scoped as a broader research programme, the aim was to use the tablet as a general purpose platform, to improve computing education and research. With a diverse group of computer science education researchers the work was firmly embedded within an existing research context, and a pedagogical framework ranging from support for the work of individuals through pairs to groups, both local and widely distributed. The notion of explanograms with such a ready at hand capture mechanism also had potential beyond the computing domain to support study in a broad range of disciplines.

The first phase of use was intended to apply the tablets as a means for capturing novice programmers' work in-situ and as it evolved. The history of the activity engaged in for specific assigned tasks was to be stored for each participating student, including an assigned question, any accompanying "doodles" [19], and a think-aloud recording of the student thought process.

In a subsequent phase in addition to supporting the BRACELet research by capturing individual student contributions, the technology would be applied to recording pair programming sessions for subsequent analysis. The usability and connectivity issues for such joint use would be the subject of study, in addition to analysis of the resulting data capturing the pair programming process.

Once fully distributed functionality for explanogram recording had been established, the next phase of the research would investigate distributed collaborative learning through peer programming or collaborative design tasks. The existing collaboration between AUT University and Uppsala University [4], would be augmented by a collaboration involving postgraduate students from the AUT University collaborative computing course.

As noted in [15] "student work within teams is a reality for managers, educators, and organizations. It is also an important value for our society. For these reasons, it is important that we determine how to teach students how to work in a global environment". Likewise the learning value of "Open Ended Group Projects" and team based learning is asserted in [8]. The teaching and learning goals of the project therefore included developing capabilities in global software development by giving students exposure to the complexities of a global software development project.

The level of funding secured to date has meant a rescoping of the project, with the initial goal being to produce a working proof- of-concept, namely a prototype working tablet platform with bare-bones connectivity to the explanogram streaming server. The subsequent phase of the project extended these goals to the production of a more robust application with greater functionality, and a limited evaluation of its use. Accompanying the application release was to be a distribution kit, installation and release package and instructions available via a project website. It was also planned to provide results from individual student use contributing to the existing corpus of data within the BRACELet study. This publication itself constitutes a proposed project deliverable with the goal of disseminating the work.

3. PROGRESS TO DATE

The initial development phase of the project was conducted over the summer break (December/February in New Zealand) of 2007/2008. During this period the team became familiar with the Microsoft tablet technologies, the digital ink APIs and Visual Studio and C# programming environment. A sample application provided by Beryl Plimmer was very helpful in understanding the operation of the pen based technologies and digital ink. Arnold Pears, the researcher who had conceived the original concept of the explanograms [10] and led the development effort in Sweden to produce the server version of the explanogram software (*http://explanogram.it.uu.se/*) visited AUT University during this period. Over the week of his visit he shared information about the server software and helped set up a working version for test purposes on an AUT server. At the end of the break an experimental prototype had been developed, which successfully demonstrated the ability to capture and replay an explanogram using the Tablet PC.

The original ANOTO pen based explanograms [11], supported both silent and sound replay versions from the explanogram streaming server, as well as the ability to upload a graphical background over which the explanogram could be replayed. The initial Tablet platform proof of concept delivered the following functionality at both Tablet (client) and server levels:

For the silent version

- Capture strokes from the pen
- Save stroke data to local file (XML file)
- Save to database (explanogram server)
- Reload the stroke from local file (XML file)
- Playback the stroke

For the sound version

- Record sound into wav format
- Convert wav to ogg format
- (still to be delivered)
    - Add time stamp to ogg file
    - Upload to server
    - Playback sound

Other controls

- Load picture to background
- (still to be delivered)
    - Load webpage
    - Resizable control

Building upon this work completed over the summer break, a team of two students undertook a capstone project to further develop the system. That work was into its second semester at the time of writing this paper. The original development plan had been conceived as figure 2 indicates, with a plan to refactor and tidy the core of the application, and produce a more tidy, robust and usable application. The ability to store background images across both Tablet and server platforms and the ability to incorporate sound were key features to be delivered. The handling of documents (e.g. explanograms layered upon interactive applications such as web browsers or IDE's) was a desired feature but highly complex (cf. [3] for a discussion of the issues related to annotating "dynamic digital documents") and considered most likely to be delivered by a subsequent team.

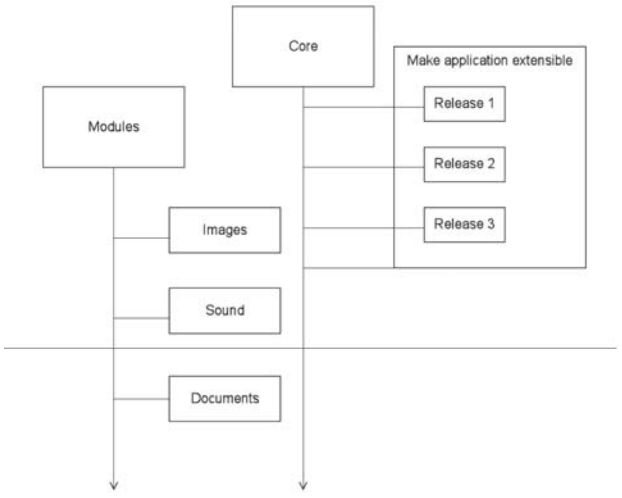


Figure 2: Development Plan for Tablet PC Based System

At the time of writing, the team had produced releases one and two of the core application from figure 2. A concrete plan for usability testing of the application had been developed, but had been put on hold while the architecture for release 3 was being revisited. The releases and the image functionality had now reached a relatively stable state, but the architecture was being refactored based upon the model view controller design pattern [7] to improve its extensibility.

Features for these two releases were:

- Revamped user interface making use of a document metaphor and which provided more feedback and status information than the original.
- New file format (based on the ZIP specification) which allows for images and associated media to be packaged into a single file.
- Improved loading and saving functionality.
- Draw and playback in the same window.
- An installer and uninstaller.
- Login to the server without editing configuration files.
- View previously uploaded drawings from the client software.
- Upload images to the server without needing to know the unique database ID.

Development of the sound module was also in progress. The plan was to have a fully functioning and tested application available for broader release and wider field testing by further researchers and educators at the conclusion of the capstone project (November 2008). At the time of writing this plan was considered quite viable.

4. DESIGN OF SYSTEM

In the design of the system the team has encountered several challenges.

4.1. Heterogeneous Technology Set

Among these challenges were simply familiarising themselves with the wide range of proprietary and open source technologies involved:

Proprietary:

- The ANOTO pen technology
- The Tablet PC technology
- Microsoft Visual Studio
- The .Net Framework
- The Tablet PC Digital Ink APIs
- Google Sites for the project repository
- VMware virtual server technology

Open Source:

- JAVA servlet programming (differing versions)
- Apache and Tomcat server technologies
- MySQL database
- PHP scripting language
- Subversion for source control
- XML data formats
- SSH tunnelling techniques for secure access

With such a diverse range of technologies, meeting the design goals of interoperability and non interference with the existing legacy application presented significant challenges for the team. Therefore the next section will briefly review the prior design and the evolution of the existing design to indicate the nature and scope of the challenges in maintaining compatibility between the differing technologies and applications involved.

4.2 Compatibility Issues

The team encountered a number of compatibility issues when endeavouring to develop the tablet PC application.

One issue was in evaluating the technology available on the Tablet PC, and ensuring the data it captures can be transferred successfully to the existing server.

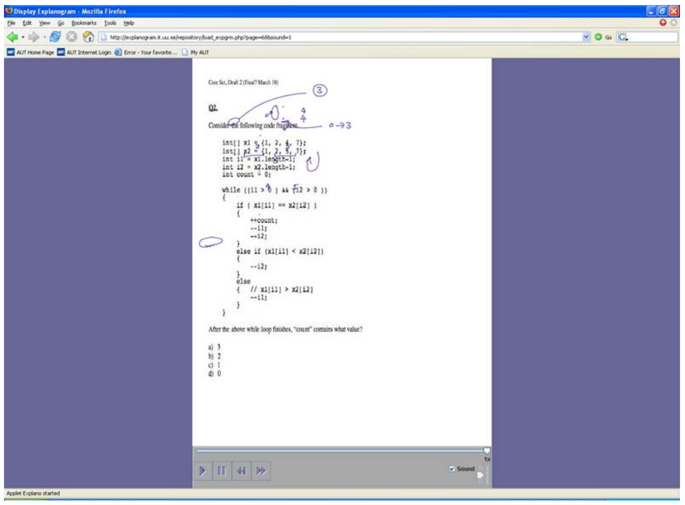
The Microsoft Ink API provides extensive functionality which can be used for anything ranging from handwriting recognition to actions based on the direction of pen strokes. However, the API did not provide time based stamping which is essential for the recording of an explanogram. For this extended properties may be attached to a Microsoft stroke object using a timer and stored timestamps in a stroke.

When this stroke data is sent to the server, it is packaged into a series of XML documents that describe the explanogram. These XML documents, along with background images or associated media, are packaged into a ZIP file. This file is then sent directly to the server where it is interpreted and stored into the MySQL database.

Given the extensive functionality available within the Microsoft Ink API, it has been tempting for the team to implement extra features especially those which are easy to implement or would improve usability. An example is deleting strokes after they have been drawn - the Microsoft Ink API easily allows for this. Because time data is associated with every stroke, it would be possible to have an explanogram which had a drawing which was visible for part of it, and was then deleted later. This is inherently different from a pen and paper based system. When using pen and paper, it is not possible to delete strokes because it is physically not possible to remove ink from the paper. The challenge therefore, is to provide the user with a good experience on the Tablet PC and allow them to be able to do things they expect from a typical application, while still maintaining compatibility with the pen system.

For now, the team has made the decision to include some of the functionality that has been easy on the tablet, such as being able to delete a stroke, while making do with the limited set of functionality on the server such that, for example, pen strokes which have been deleted on the tablet are visible throughout the entire drawing when displayed on the server. From a usability standpoint, it will be necessary at some point to ensure that the server representation of the drawing and the tablet representation of it are consistent. At this prototype stage, the team felt it was important to ensure usability of the tablet application matched what a user would expect from a typical Windows application.

With two separate input devices sending two completely different sets of data to the server, two separate modules for receiving data are presently demanded. At the beginning of this year, the team had a decision to make concerning how best to develop these two modules. There are problems in attempting to use the existing pen data receiver on the server and adapting it for the data which the Tablet PC can send. While it can receive pen strokes and time - which is what the pen provides - it cannot receive additional data which the server nevertheless supports on playback. One example is a background image.


Display Explanogram - Mozilla Firefox
Core Set, Draft 2 (Final? March 18)
Q2.
③
0 → 3
Sound

Consider the following code fragment.

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;

while ((i1 > 0 ) && (i2 > 0 ))
{
    if ( x1[i1] == x2[i2] )
    {
        ++count;
        --i1;
        --i2;
    }
    else if (x1[i1] < x2[i2])
    {
        --i2;
    }
    else
    {   // x1[i1] > x2[i2]
        --i1;
    }
}
```

After the above while loop finishes, "count" contains what value?

a) 3
b) 2
c) 1
d) 0

Figure 3: Web Based Explanogram with Background Image

As depicted in figure 3 above, (where a programming examination question as a background graphic has been annotated) it is possible on the server side to store a background image which will be displayed in the background of an explanogram while it is being played. However the user process for doing this with the pen is extremely difficult – as the server application is still somewhat experimental the user must discover what the database's unique ID for the explanogram is, and then manually upload the background image they require to a specific directory on the server.

From the team's point of view this was not ideal given that the user would be able to insert a background image easily from the Tablet PC application. Unfortunately the existing server code did not have any ability to receive images and store these on to the server. One option could have been to substantially rework the architecture of the existing code in order that it could receive extra information like background images. However it was vital that the new development did not affect the functioning of the pen based server code. Therefore, it was decided that additional code would be developed independently to sit alongside the existing code. At a suitable time the shared pen and tablet code could then be consolidated into modules.

The existing pen functionality is written in Java within Tomcat, while the new Tablet code is written in PHP. Both translate the incoming information and store it into a common database, such that one Java player on the server is able to play back both sets of drawings, which may then be displayed in a single location.

A further challenge was posed by the incorporation of sound into the Tablet based explanograms. The original system's support for sound used the OGG file format with an offset from the start of file being used to support time synchronisation when sound was replayed with the text content of the explanogram. The Tablet PC provides native support for Microsoft .Wav file formats, which would enable an easy implementation of the desired functionality on the Tablet but poor compatibility with the server features. Thus the development has been inherently constrained by the need to remain compatible with a functioning legacy application on the server, while tempted by the ability of the Tablet PC to offer a superset of the existing functionality. Figure 4 depicts the existing explanogram system Architecture.

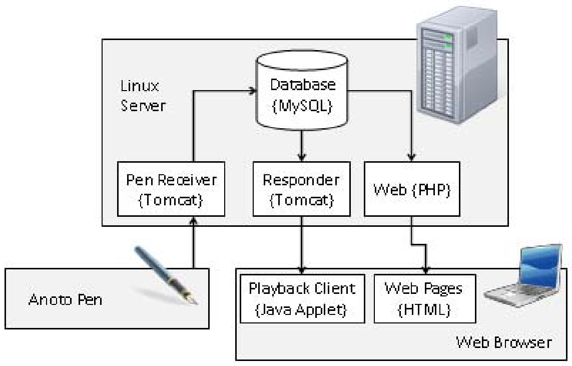


Figure 4: Previous Explanogram System Architecture

The Tablet PC based application now augments the prior architecture with the elements shown in figure 5 below.

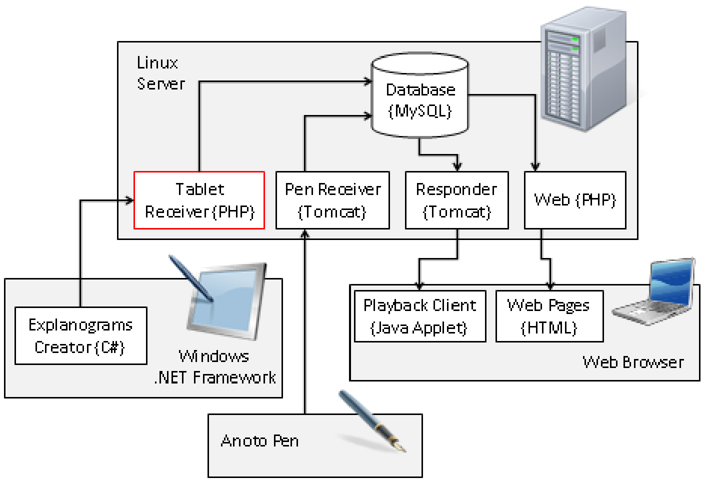


Figure 5: Adapted Tablet PC Based Explanogram System Architecture

The internal architecture of the application is still evolving, with the team engaged in a progressive refactoring process in order to produce an extensible and robust internal design. A sample explanogram is portrayed in figure 6 below indicating the user interface developed for the tablet PC. As can be seen a drawing palette enables pen colours to be selected, and the explanogram can be saved and replayed locally or uploaded to the server

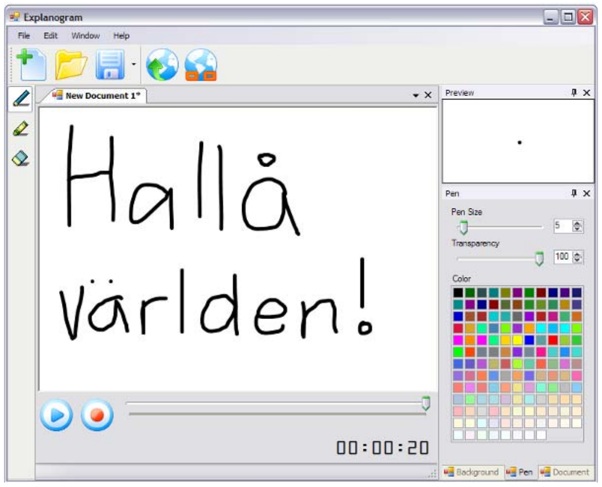


Figure 6: User interface for Tablet PC Based Explanogram Client

4.3 Intellectual Property Issues

Another constraining dimension of the design has been the question of ownership of the elements of the work. For the existing explanogram application, ownership rested with Arnold Pears. The terms of the Microsoft grant were that the Tablet PC component would be made available freely for others to use, including Microsoft. This placed parameters around the design of the different components and required that the team ensure a cleanly separated interface at the Tablet level.

5. DESIGN PROCESS

The methodology selected for the capstone project combined practices and processes from three methodologies – the Design Science Research Process, the Star model and Extreme Programming. The Design Science model was created to model the design process one follows when identifying a problem and then translating this problem into a potential solution followed by eventual academic publication. The Design Science model specifically focuses on ensuring a consistent way for researchers to carry out the process and therefore a way to recognise this research as having followed a thorough process [12].

The Star model is a user centred design model that was developed by Hartson and Hix [13] and is based on modelling HCI design practices. The key to this model is evaluation after every step and the ability to enter the cycle at any stage. The evaluation comes in the form of review by experts and surveys conducted with users. The idea is to create a more 'bottom-up' approach encompassing practices such as prototyping rather than the traditional 'top-down' waterfall style methodologies.

Extreme Programming is an agile methodology which encourages client-user-developer collaboration and the idea that change is to be expected and embraced. It was created specifically for small teams of people and provides processes which aim to reduce the cost of changes coming from vague requirements [2]. Given the small team and vague requirements that were elements of this project the team felt that XP was a perfect fit. Combining these three methodologies to produce a strategy appropriate for this project generates the following graphical representation.

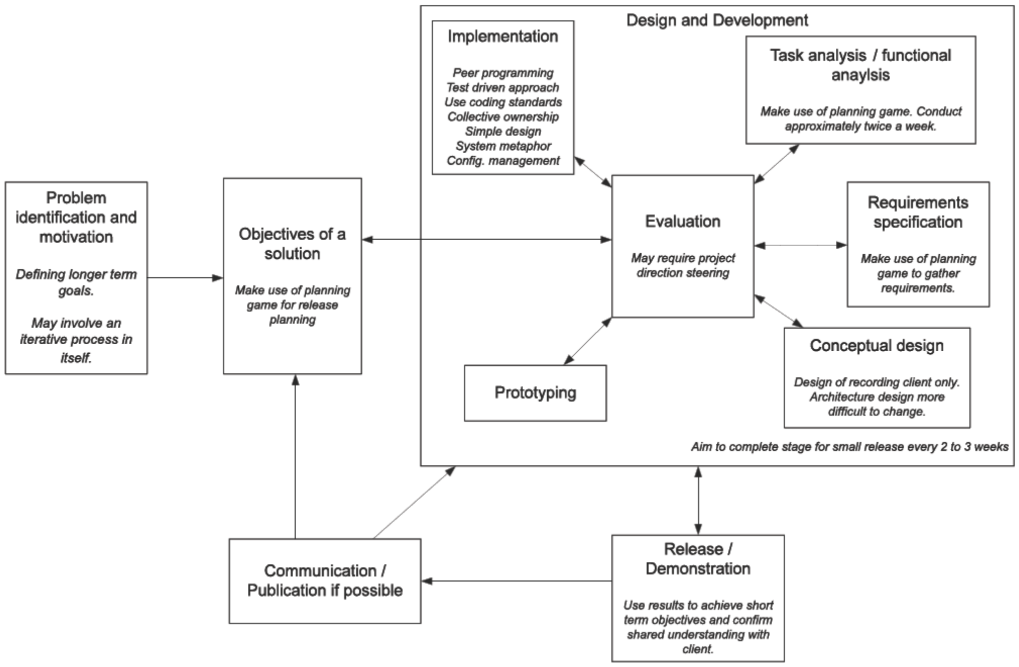

Design and Development
Implementation
Peer programming
Test driven approach
Use coding standards
Collective ownership
Simple design
System metaphor
Config. management
Task analysis / functional anaylsis
Make use of planning game. Conduct approximately twice a week.
Problem identification and motivation
Defining longer term goals.
May involve an iterative process in itself.
Objectives of a solution
Make use of planning game for release planning
Evaluation
May require project direction steering
Requirements specification
Make use of planning game to gather requirements.
Conceptual design
Design of recording client only. Architecture design more difficult to change.
Prototyping
Aim to complete stage for small release every 2 to 3 weeks
Communication / Publication if possible
Release / Demonstration
Use results to achieve short term objectives and confirm shared understanding with client.

Figure 7: Research Methodology and Evaluation Process

This approach takes the Design Science model and uses it as a framework for the entire methodology. The *problem identification and motivation* stage allows for longer term project goals to be defined. The *objectives of a solution* stage allows for a release to be planned by thinking about why this release is required and an overview of what is expected. The *design and development* stage allows for actual development and artefact production to take place. The *release/demonstration* stage involves the development work being packaged for release to users or demonstration to the client [2]. At this point it is entirely possibly to move back to *development* based on results of user or client feedback. Finally, the model allows for *communication* of the results of *development* and *evaluation* by way of paper publication or other means. After this point, the model allows for the cycle to be repeated to create more release objectives.

The Star model is integrated mainly into the design and development stage of the overall model. The Star model allows for entry into any point of the star, each of which should be followed by evaluation. The requirements specification stage allows for requirements to be gathered from the client or based on analysis arising from the already completed objectives stage. The conceptual design stage allows for the overall design to be modelled and considered. The prototyping stage allows for prototypes to be developed based on expert analysis or already completed evaluation. Finally, the implementation stage allows generated ideas to be put into actual code or other artefacts [13].

Extreme programming processes are used within the context of other parts of the defined process. The planning game, for example, is used at various points to help describe requirements. This makes use of such techniques as user stories to complete the process. Other practices are used at the implementation stage such as pair programming, which has two developers working at one computer; shared ownership, giving any developer the right to change any code and configuration management, which ensures that the same version of the code is being worked on at any given moment [2].

5.1 Reflections on Actual Design Process

In carrying out the development process, as student developers we found that the planned process provided a good basis for development and largely was followed – even when working in an organic manner we found that the process was relatively natural and fitted in well with our work processes. On the occasions where we deviated from the planned process we often found we had problems which we would not have had if we had understood and followed the process better.

The design and development stage of the process tended to form the bulk of activities, and here we found that our ad hoc and organic processes actually conformed closely to those planned. For example, our process of gathering requirements and analysing required functionality fitted well into the Requirements specification and Task analysis / functional analysis stages. Our natural tendency to produce prototypes of our software as we worked fitted well into the Prototype stage. We did end up effectively 'throwing away' some of these prototypes, but this was not a problem and allowed us to evaluate our progress and make improvement. The process allows for all of these.

We conducted a number of formal releases, accompanying each with release notes and a relatively stable copy of the application code, thereby fulfilling the Release / Demonstration stage. We are, of course, communicating the results of our efforts within this paper, fulfilling the Communication stage.

As an example of what occurred when the process was not adhered to, the team initially failed to fully observe the separation of the *Problem identification and motivation* ("problem stage") and *Objectives of a solution* stages ("objectives stage"). After the first client meeting we began preparing for what we expected would be the final software features. The problem encountered was that there was a lack of understanding of the bigger picture about *why* the project was being perused – we were immediately concerning ourselves with *what* and *how*. This was probably a problem of not fully understanding our own planned process – we needed to appreciate that the problem stage really does not concern what the project will produce, rather it concerns the ultimate high level goals of the project, in this case the goal of aiding novice software developers.

On reflection, it is almost a surprise that the process was such a success. This is because our past experience has been that after a process is devised, it is almost natural to have a desire to work in an ad hoc manner not prescribed by the process, probably caused by an eagerness to produce results and a perceived overhead of the prescribed process. On this occasion, we had a well planned process which we actually expected to model reality. This meant that even without specifically meaning to, we were able to naturally follow it. On the occasions when we did not properly follow our own process, we achieved better results by stepping back and ensuring we properly understood and followed the prescribed steps.

5.2 Research Linked Teaching & Learning

A research linked model of teaching and learning imposes additional methodological requirements. The methodology of figure 7 incorporates the development, the usability evaluation and the research components of the project, but omits the aspects related to research ethics approval. AUT University mandates such approval in order to publish this form of work as "research", rather than simply conducting it as a 'secret' teaching and learning activity. The participation of students as co-authors of this paper is an inherent aspect of the learning desired for their capstone project, which consistent with [4] we consider an excellent model for fostering undergraduate research. Therefore ethics approval has been gained for two aspects of the project 1) developing the software and usability evaluation with other students as research subjects, 2) evaluation of the research process itself and its historical development by the participants.

6. GLOBAL DIMENSIONS

On reflection, the chosen design process had omitted to fully consider the global and distributed nature of the project.

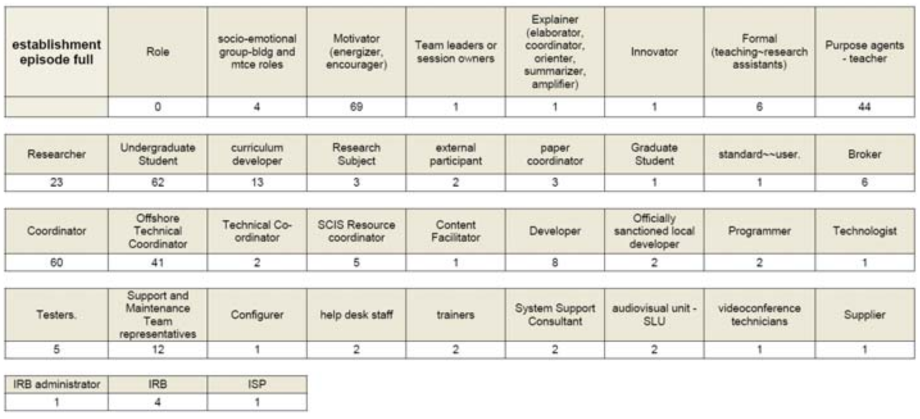
6.1 Diversity of Participant Roles

One aspect of the global context required that both the Principals in this research, and the capstone project supervisors assume a number of coordination and facilitation roles on behalf of the team. In hindsight it may have been better to more explicitly define these roles. For instance the roles identified by the capstone project team members were the following:

- Client communication coordinator
- Software Developer
- Project Manager
- Usability Expert
- UI Designer

Yet in practice the full research team have also had to perform or to interact with several additional roles. For instance a review of the number and variety of roles involved in one "episode" (phase) of a global virtual collaboration [6] presents the rich list of actors below:

Space precludes full discussion of this topic, but the innate challenges of global virtual collaboration have been noted in [5].



establishment episode full	Role	socio-emotional group-bldg and mtce roles	Motivator (energizer, encourager)	Team leaders or session owners	Explainer (elaborator, coordinator, orienter, summarizer, amplifier)	Innovator	Formal (teaching~research assistants)	Purpose agents - teacher
	0	4	69	1	1	1	6	44

Researcher	Undergraduate Student	curriculum developer	Research Subject	external participant	paper coordinator	Graduate Student	standard~~user.	Broker
23	62	13	3	2	3	1	1	6

Coordinator	Offshore Technical Coordinator	Technical Co-ordinator	SCIS Resource coordinator	Content Facilitator	Developer	Officially sanctioned local developer	Programmer	Technologist
60	41	2	5	1	8	2	2	1

Testers.	Support and Maintenance Team representatives	Configurer	help desk staff	trainers	System Support Consultant	audiovisual unit - SLU	videoconference technicians	Supplier
5	12	1	2	2	2	2	1	1

IRB administrator	IRB	ISP
1	4	1

Table 6.14: Establishment Episode Full – Coded 'Roles'

Figure 8: Roles in a Global Virtual Collaboration [ex. 6 p. 211]

6.2 Rolling Cast of Actors – Team Turnover

For differing reasons, (e.g. cessation of funding for research assistants over the initial development period, brief onsite visits of remote team members, school restructuring etc.) the people who have been involved in the project at any given time have varied. As people have left the project, they have often taken with them knowledge of various aspects – both technical and procedural. As new people have been brought in, their skill sets have helped to shape the focus of the project. There have been successive handovers and information loss, and familiarisation issues with which the team members have had to contend, not unlike a true commercial development context [16].

For example, the supervisor for the student developers towards the beginning of the year was a human computer interaction expert. As such, effort was put into ensuring that the application was as usable as possible and progress was able to be reviewed successfully by the supervisor. As she has now left the university, a new supervisor has been appointed for the team whose expertise lies in software development and system architecture. This has shifted the focus more from usability to ensuring that the class structure of the application is appropriate to ensure extensibility and compatibility.

At times, the loss of people not even directly related to the project has caused delays. As is explored within section 6.4, the team desired to use Subversion (SVN) to track the configuration management and ended up setting up their own SVN server. In part, this was because the administrator for the School's CVS server had promised to install SVN as well, but left the university shortly after this.

6.3 Communication Processes

Communication in a global software development project can be challenging [5, 6, 15, 17]. At the project initiation we arranged a videoconference session over the Karen access grid network [5] from AUT University to Arnold Pears at Uppsala. While eventually successful this was a highly fraught one-off exercise, similar to those noted in [5]. Arnold visited the team for a week in December of 2007 so that they could work together face to face – a crucial visit which enabled us to share information to familiarise with the still somewhat experimental system and set up a test server on AUT premises. Subsequent communications have seen a combination of Skype sessions and email in use, with the local team, supervisor and on-site sponsors at AUT also regularly meeting face to face.

In a university setting – or indeed in any large organisation – there are often difficulties getting certain systems in place to support development and testing. The team has encountered these issues.

Toward the beginning of the project, the student developers chose to make use of Google Sites, a wiki-like service provided by Google which allows the creation of interlinked pages while keeping the full revision history of all documents and files. The students signed up to the service using their AUT email addresses. Afterwards it was noticed that the use of these email addresses meant that AUT administrators would be able to take control of the Google Site, and potentially delete it or modify access. Communication with the administrators was not straightforward for the student team, as the team had to work by proxy through the technology resource coordinator for the school. In addition, there was difficulty in explaining the exact nature of the risk and what the university administrators could do about it.

The eventual solution was that the AUT Technology Services took control of the Site, and guaranteed access by the relevant stakeholders by putting the administrator password into a 'digital safe' with a note that the Site should not be deleted under any circumstances.

Security concerns were another issue that arose on various occasions. The team has a number of hardware devices which it uses – a number of Tablet PCs and a physical workstation. The university network allows only computer systems whose MAC addresses are known onto the network. It was therefore necessary to communicate the Tablet PCs MAC addresses to the university Technology Services, as well as the MAC addresses for virtual machines the team was using.

6.4 Software Config. & Test Environments

The original server software developed by students at Uppsala University was functional, but difficult to install and run. The team at AUT having not been fully involved in the development of this software were not aware of the exact procedure for setting up this software, nor the versions which were available. As such, when the current student developers took control of the project they simply copied the existing server code which was already on one of the Tablet PCs – this code did not necessarily represent the cleanest or most up to date version of the server.

The decision was made to package this server code into a working standalone server rather than running it concurrently on development computers as had been done. The development computers ran Windows but the server itself originally ran on Linux. Linux was chosen as the platform for the standalone server as this would allow for distribution of the entire server within a virtual machine without licensing issues. The process for packaging the server was more difficult than expected, with the Tomcat components of the server having difficulty communicating with the MySQL database. On a whim, the team tried running the Tomcat components upon the Windows version of Tomcat on Linux using the WINE Windows compatibility layer. Surprisingly this worked and has been stable since.

The server has since been packaged into a virtual machine for ease of deployment onto demonstration machines. It runs within the free VMware player and allows one to demonstrate the server-client connectivity without network access. The free VMware Server is used to run an instance of the server which operates full time and is accessible from within AUT. The use of VMware Server allows for snapshots to be made of the server for easy rollback if something goes wrong, and for the server to run concurrently with an SVN server and a workstation on the Team's single physical computer.

Configuration management for the software is made difficult by the various technologies which must co-exist and the different environments used in the past. The original server components were developed in Java, and made use of CVS as a source control repository. The Tablet PC application, however, is developed in C# with Visual Studio. While it would have been possible to use CVS for this, it is not possible to integrate CVS particularly well with Visual Studio. There does however, exist a very good plug-in for SVN which works well with Visual Studio. For this reason SVN is used when developing the Tablet PC application.

There was some difficulty in setting up the SVN server, due to the service not being offered on a server accessible from outside the university – the server available offers only CVS support. The team was able to work around this by creating their own SVN server (again running within VMware Server and allowing access to it by creating an SSH tunnel into the university.

SSH tunnels are used to allow access to a number of different services within the university and allow the team to access these services without having to go through the fraught process of seeking and obtaining permission from the university Technology Services department to make these servers accessible from outside.

The process for establishing the SSH tunnel typically involves the generation of a public-private key pair, the private component of which is placed onto the university's SSH server while the public key is made available to those desiring access. The SSH server is configured to allow those with the relevant public key only to tunnel into the relevant server; no other SSH access is granted. The user desiring access then uses a Windows Plink.exe command to connect to the server with the public key. They access the relevant services at the 'localhost' address and can disconnect afterwards, all while still respecting the university firewall.

7. FUTURE WORK

It is planned to make the software freely available to collaborating parties with shared interests and Tablet PC rich environments. The team are working on setting up a repository from which the application can be downloaded for more general use, once it reaches a stable release point by the end of the semester, and a more general solution for hosting the server is established.

As has been noted, there are currently issues with attempting to access the server from outside AUT university. The solution is to make this available without having to create a tunnel. Securing permission from the university Technology Services and conducting a security review of the server is currently in progress.

It is anticipated that much of the server code which was created could be consolidated – having a server which uses either Tomcat or PHP, but not both, is likely to ease deployment and maintainability. One future solution could be to rework the Tomcat modules such that common functionality is extracted. The decision to use PHP at this early stage, is because PHP is a scripting language allowing for easier prototyping and bypassing of the legacy code.

In the future it may be necessary to replace a great number of the components of the existing server if the extended functionality of the Tablet PC is desired to be included. The Java applet player on the server, for example, may need to have significant new functionality added in order to understand such concepts as deleting strokes after their insertion. The database too, will need to have more information able to be added if this extended functionality is to be recorded. In some cases this could be as simple as adding new columns to database tables. Of course the team is interested in ensuring that the pen continues to work so modifications will have to be made with care.

The decisions on exactly how to modify the server to be more compatible with the tablet have not yet been made. However, as the project moves forward, it is expected that future student developers working on the project will be able to take some of the decisions made by the current developers and use these to form a sound strategy on the best way to proceed.

8. CONCLUSION

This report on the phases of a distributed software development project developing explanograms for Tablet PC's has reviewed both product and process elements within a research-linked model of teaching and learning. The design of the system has been outlined together with the key challenges and design constraints which the current student development team has recently encountered and the research and development methodology employed. Some of the technical challenges have been expressed at quite a micro-level, but they do realistically reflect the student experience. A legacy pen based application for providing replayable snippets of explanation (explanograms) has been augmented to interoperate with a Tablet PC application providing a superset of the previous pen based system's functionality. The multiple challenges of a global software development project and how the team has addressed them have been reviewed in the context of this project. Blending technologies and dealing with conflicting intellectual property are messy, much like many business applications. This messiness, in itself has been a valuable part of the learning experience for the students. The design and evolving functionality of the Tablet PC based explanogram application has been described. A progressive usability process and plans for evaluating the software have been presented. Plans have been outlined for extending the work and for making a trial version of the Tablet based software communicating effectively with an open server freely available for a wider group of educators and researchers by the end of 2008. It is hoped that others will adopt and use the software to help develop the functionality, validate its usability and demonstrate the value of explanograms in supporting teaching and learning processes. While computing educators are seen as the likely first users of the application, it has the potential to serve educators and students in many other disciplines.

9. ACKNOWLEDGMENTS

This work has been enabled by the provision of a project grant from the Microsoft Research Asia - 2007 Mobile Computing in Education Theme, and by the donation of a tablet PC from Cyclone Computers. The contributions of the student developers (Tang Dang Bao Trung, Jonathan Hill, Yong Liu) and their supervisors at AUT University (Doug MacKenzie, Diana Kassabova, Anne Philpott) have been key to the progress of this work. We acknowledge the initial provision of sample tablet PC code by Beryl Plimmer of the University of Auckland, which greatly assisted the early design work.

10. REFERENCES

[1] Arnold, D. Editorial for Inaugural Issue of JOCCH: Pasteur’s Quadrant: Cultural Heritage as Inspiration for Basic Research in Computer Science. *ACM Journal on Computing and Cultural Heritage (JOCCH) 1*(1). 1:1 - 1:13.

[2] Beck, K. *Extreme programming explained*. Addison Wesley Longman, Reading, 2000.

[3] Chen, X. and Plimmer, B., Code Annotator: Digital Ink Annotation Within Eclipse. in *OZCHI 2007 Proceedings*, (Adelaide, Australia, 2007), CHISIG.

[4] Clear, T. and Kassabova, D. A Course in Collaborative Computing: Collaborative Learning and Research with a Global Perspective. in Guzdial, M. and Fitzgerald, S. eds. *Proceedings of the 39th ACM Technical Symposium on Computer Science Education*, ACM, Portland, Oregon, 2008, 63-67.

[5] Clear, T. Global Collaboration in Course Delivery: Are We There Yet? *SIGCSE Bulletin, 40* (2). 11-12.

[6] Clear, T. Supporting the Work of Global Virtual Teams: The Role of Technology-Use Mediation *Computing and Mathematical Sciences* Auckland University of Technology, Auckland, (submitted for examination), 1-778.

[7] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] Hauer, A. and Daniels, M. A learning theory perspective on running open ended group projects (OEGPs). in Simon and Hamilton, M. eds. *Conferences in Research and Practice in Information Technology*, ACS, Wollongong, NSW, Australia, 2008, 85-92.

[9] Manford, C. The impact of the SaaS model of software delivery. in Mann, S. and Lopez, M. eds. *Proceedings of the 21st Annual NACCQ Conference*, NACCQ, Auckland, New Zealand, 2008, 283-286.

[10] Pears, A. Enriching Online Learning Resources with Explanograms *International Symposium on Information and Communication Technologies (ISICT'03)*, Dublin, Ireland, 2003

[11] Pears, A. Explanograms: Low Overhead Multi-media Learning Resources. in Korhonen, A. and Malmi, L. eds. *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science, 2004, 67-74

[12] Peffers, K., Tuunanen, T., Gengler, C., Rossi, M., Hui, W., Virtanen, V. and Bragge, J., The Design Science Research Process: A Model For Producing And Presenting Information Systems Research. in *First International Conference on Design Science Research in Information Systems and Technology (DESRIST 2006)*, (Claremont, CA. Retrieved 17/05/2006 from http://ncl.cgu.edu/designconference/DESRIST%202006%20 Proceedings/4A_2.pdf, 2006).

[13] Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S. and Carey, T. *Human-Computer Interaction: Concepts And Design*. Addison & Wesley, Reading., 1994.

[14] Raymond, E. The Cathedral and the Bazaar. *First Monday, 3* (3). Retrieved 16 Apr 2006 from http://www.firstmonday.org/issues/issue2003_2003/raymond /

[15] Richardson, I., Milewski, A., Keil, P. and Mullick, N., Distributed Development - An Education Perspective on the Global Studio Project. in *28th International Conference on Software Engineering (ICSE'06)*, (Shanghai, China, 2006), ACM, 679-684.

[16] Sim, S. and Holt, R., The Ramp-up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. in *Proceedings of the 1998 (20th) International Conference on Software Engineering*, (Kyoto, Japan, 1998), IEEE.

[17] Swigger, K., Brazile, R., Harrington, B., Peng, X. and Apaslan, F. Teaching Students How to Work in Global Software Development Environments *International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2006 (CollaborateCom 2006)*, IEEE, Atlanta, Georgia, USA, 2006.

[18] Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. and Prasad, C. *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. Conferences in Research and Practice in Information Technology, 52*. 243-252

[19] Whalley, J., Prasad, C. and Kumar, P. *Decoding Doodles: Novice Programmers and Their Annotations. Conferences in Research and Practice in Information Technology, 66*. 171-178.

Implementing a Contextualized IT Curriculum: Ambitions and Ambiguities

Matti Tedre
Tumaini University
Iringa University College
B.Sc Program in IT
Iringa, Tanzania

Fredrick D. Ngumbuke
Helsinki Metropolia University of Applied Sciences
Helsinki, Finland

firstname.lastname@acm.org

Nicholas Bangu
Tumaini University
Iringa University College
Iringa, Tanzania

Erkki Sutinen
University of Joensuu
Dept. of Computer Science and Statistics
Joensuu, Finland

ABSTRACT

In this article we report the combined findings from an ethnographic field study and action research on implementation of a newly founded IT program in rural Tanzania. We have found that the competences and skills of IT professionals in developing countries differ from the competences and skills of IT professionals in industrialized countries. Also workable pedagogical approaches, students' educational backgrounds, teachers' level of education, attitudes towards university education, aims of education, organizational and administrative frameworks, and people's motivations differ between industrialized and developing countries. We report some ways in which developers of functioning, sustainable, relevant, and motivating IT programs in developing countries face different challenges than their industrialized-world counterparts. We finish this paper with a number of lessons learned that we hope to be useful for other people undertaking similar projects in the developing world.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education, curriculum*

1. INTRODUCTION

In 1965 the Ministry of Finance in Tanzania began the first information technology (IT) program in Tanzania [27]. By 1974 there were seven computers in the country. The introduction of these computers as well as the whole idea of computerization faced some problems, such as the lack of qualified Tanzanian IT personnel, uncoordinated IT project planning, and failed government interventions [27]. The University of Dar-es-Salaam was the first to start information technology teaching initiatives—yet principally, their whole range of courses was not aimed at training people to become specialists on a wide range of computer skills, but rather at providing them with computer appreciation or with skills on how to solve problems in a specialized area [27]. In 1974 the University of Dar-es-Salaam took a major step to start a M.Sc program, which, however, did not work well, because there was no other institution that could offer a one- or two-year diploma or certificate in the field [27]. That initiative of the University of Dar-es-Salaam faced great problems and was finished in 1984. As a result of those problems with the M.Sc program, the University of Dar-es-Salaam started a diploma program instead [27]. Most of the programs launched in Tanzania have been copied directly from the Western curricula.

In Tanzania the imported, theoretically oriented computing curricula have not met the expectations. That same problem seems endemic to the country's educational system: several universities offer theoretical education in fields such as agriculture and civil engineering, yet no major improvements of agriculture have been made since the 1960s, and most engineering project bids today are won by Asian or European firms. This has led us to believe that in addition to theoretical knowledge, Tanzanian higher education should focus on developing students' *skills.* Students should not only know *what* to do but also *how* to do it.

The demands of IT education in Tanzania differ quite a lot from the demands of IT education in industrialized countries. Many of those differences derive from different needs of the society and different aims of education, whereas many others derive from differences in educational background of students and staff, and, most importantly, differences in environment (including cultural, social, political, natural, economic, and all other kinds of environment.) Most remarkably from the viewpoint of IT education, in Tanzania experts are not needed as much as all-round IT bricoleurs are.

Tumaini University launched a B.Sc program in IT in September 2007. A significant amount of work was done in order to refine the curriculum to contextualize it to fit the Tanzanian context [8, 19, 26, 33, 40, 41, 42]. Tumaini's B.Sc program in IT is based on continuous, research-based, formative program development. In this paper we present and analyze the challenges we faced during the first year of implementation of Tumaini's B.Sc program in IT (hereafter *BIT program*). Our primary aim is to elucidate on our increased understanding of the principles of a contextualized IT program. Our secondary aim is to convey knowledge about pitfalls, which may seem obvious from the perspectives of other disciplines and which we were aware of from the very beginning, but which we still had to learn the hard way.

2. RESEARCH METHOD

This paper reports an investigation of the educational, social, and cultural environment of a new educational program. As a combination of ethnographic research and participatory action research [6], this research focuses on exploring challenges and prospects that the Tanzanian context brings into the development of an IT program. Nuances of sociocultural interactions as well as many ethnographic observations are largely excluded due to the limited length of the paper. Typical of ethnographic research [5], we aim at exploring local particulars, emphasize adaptability in the course of study, develop new concepts over the course of the study, and represent data mostly in natural language.

Reports of interpretive research are easily biased by personal opinions and positions [16]. Therefore, it is important to elucidate our positions in the organization we study. This paper was written from the viewpoint of four people associated with the program: we write this paper *qua* associate professor (head of the program), ICT director, adjunct professor of the program, and provost (CEO) of the university. The head of BIT program was hired from outside the organization to run the program and to launch a continuous improvement process within the program. The ICT director—who is a Tanzanian citizen but has received his B.Tech degree in Europe—has worked with the university since 2004, focusing on IT support and infrastructure development, and he played a role in the design and implementation of the BIT program. The adjunct professor, who is a professor and head of computer science department in a European educational institution, has collaborated with Tumaini University in terms of IT education development for approximately ten years, and now has an advisory role in the program. The fourth author, who is the provost of Tumaini, initiated the BIT program and chaired the design and implementation process of the program.

Our views about the program's development are necessarily biased by our positions and history with the program. In this paper we present the challenges and prospects as we see them, and our views surely differ from those of other stakeholders. To complement our lived experiences we use data collected from various sources: emails, student feedback, internal memos, seminar presentations, notes on discussions, and field notes of a researcher who is conducting an ethnographic study on the development of the program. Those data sources are indicated in footnotes where appropriate. The data analysis was basic qualitative data analysis where emerging themes, patterns, and signals were analyzed for resonance with research literature [31].

Organization and Staff

Other educational programs at Tumaini fit well under Tumaini's four faculties: Faculty of Theology, Faculty of Law, Faculty of Arts and Social Sciences, and Faculty of Business and Economics. Information technology program did not fit well under any of the existing faculties, so an ICT directorate (an independent unit of a smaller size than faculty) was founded to accommodate the program. The current organizational situation of the program is illustrated in Figure 1. The ICT Directorate is divided to two: *IT support* and *B.Sc Program in IT.* ICT director is in charge of IT support, and, together with the head of the B.Sc Program, of the BIT program. Technicians can be used to assist the BIT program. During the academic year 2007–2008, teaching staff consisted of three tutorial assistants, ICT director, and one associate professor. The associate professor held a doctoral degree in computer science, whereas the other teaching staff members held B.Sc or B.Tech degrees in computing.

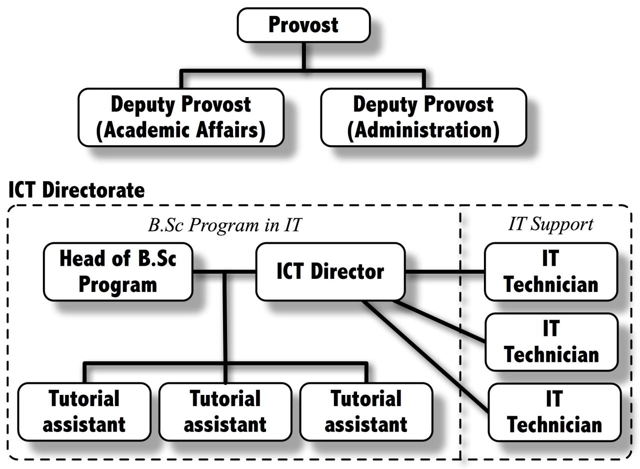


Figure 1: IT Program's Place in Tumaini's Organization

3. AMBITIONS AND AMBIGUITIES

Tumaini's original IT curriculum was based on six ambitious principles [8], which were designed to promote teaching that was 1) oriented strongly towards practice and activities, 2) based on problem-solving projects (problem orientation), 3) sensitive to context, 4) interdisciplinary in design and implementation (computer science, information technology, and computer engineering being the incorporated fields of study), 5) of a standard that would be internationally recognized, and 6) based on research. The six principles seemed obvious in the design phase of the program [8], yet the content and implications of those principles were not analyzed further. In this section we describe the ambiguities that arose in the implementation phase of the program, and we analyze the program principles further.

3.1 Principles

Tumaini's IT program was founded on six principles [8]:

- Practicality
- Problem-based orientation
- Context-sensitivity
- Interdisciplinarity
- International recognition
- Basis on research

The first founding principle of Tumaini's IT curriculum, *context-sensitivity*, refers to the idea that each society, climate, environment, economy, language, and culture pose some unique challenges for IT professionals and educators; and to the idea that local IT curricula and pedagogy should respond to those challenges. The second principle, *problem-based orientation*, refers to the typical constructivist approach of problem-based and project-based learning—that is, students work on projects that concern authentic problems and reflect on the experiences they gain while working on those projects [20, 21].

The third principle, *practicality*, refers to the idea that in Tanzania, IT professionals face all kinds of practical and theoretical challenges, and without practical training BIT program graduates may not be able to work with the various hands-on tasks that they are expected to. The fourth principle, *interdisciplinarity*, refers to the need to incorporate subject areas such as development studies and business in the IT degree curriculum, as well as to the combination of information technology (IT) curricula, computer science (CS) curricula, and computer engineering (CE) curricula. In addition, IT professionals must have the ability to work with people from various fields.

The fifth principle, *international recognition*, refers to the fact that in order for students to work in countries other than Tanzania, and in order for students to continue their studies in international master's level programs, the BIT program must be aligned with the international standards of IEEE and ACM [3, 4]. The sixth principle, *basis on research*, refers to the need to continuously develop the curriculum further, and to base any revisions of the curriculum on rigorous research.

In the design phase of the BIT program, 2006–2007, we set the principles above in a clear manner, yet the actual content of those principles and the role of those principles in implementation of the program were never clearly stated. That is, the program syllabus and curriculum guidelines explicitly referred to the six principles above, but we had to learn how to actually implement and follow them in the program. This turned out to be a much bigger challenge than we thought it would be.

Most importantly, we did not consider the fact that not everyone shares the same ideas about concepts that are very abstract (e.g., context sensitivity, problem-based learning, and practicality). That is, it was implicitly assumed that tutorial assistants, Tanzanian students, program designers, program management, university administration, and all other stakeholders would all share the same idea about the essence, importance, and implications of, say, 'practical orientation.' That implicit assumption indeed ran against the very idea of contextual understanding. In fact, students had their idea of what 'practical' means, tutorial assistants (graduates of a theoretical program in the University of Dar es Salaam) had their idea of what 'practical' means, and surely program designers, university management, and other stakeholders all had their own views on what 'practical' actually means.

We are currently on our way to clear some of the ambiguities of the abstract principles underlying the BIT program. Certainly, there will always be ambiguity concerning concepts (e.g. [43, pgs. §64–§67]), but some common understanding and agreement is necessary in order to implement, discuss, analyze, and evaluate the program. In the following sections we analyze the rationales, implications, and pitfalls of the six program principles.

3.2 Practicality

Not only is the concept of practicality ambiguous, but the tension between the principles of international recognition and practicality posed some special challenges to curriculum design. Yet the problem is not only Tumaini's: the tension between theory and practice has driven and haunted computing disciplines ever since the birth of modern computing [34, pp.283–286]. Theory is indeed the bedrock of academic computing—any academic curriculum in IT must have a sound theoretical base [14]. But there again, without design and implementation of technological tools, computing would be just idle speculation [18]. Although all IT professionals must know some theory of computing, they should not forget about the users and their real problems [12].

During the whole first academic year, we heard students demanding for more practical tasks and hands-on learning; and we heard teachers emphasizing the importance of abstract concepts and lecture-based teaching. For instance, in a number of 3-hour per week courses that were originally meant to be 2/3 hands-on and 1/3 lecture-based, it turned out that the teachers wanted to spend the whole three hours a week on lectures and give students some homework as "practicals [sic]".[1] Students' and program management's demands of practical sessions were not implemented. In staff meetings it turned out that all the tutorial assistants had, in their studies, been taught through an instructivist pedagogy, which is the prominent pedagogy in Tanzanian educational system. We finally ended up solving this issue by dividing all classes in timetable so that 1 hour per week takes place in a lecture hall and 2 hours take place in a computer laboratory where lecturing would be very impractical. This organization, we hope, brings orientation to teachers and gives some leverage to students' pleas for practical sessions. Teaching arrangements and the very environment of teaching now steers classes to a clear division to practical and theoretical sessions.

Donald Knuth [23] noted that "*Theory and practice are not just two sides of the same coin. They deserve to be mixed and blended, but sometimes they also need to be pure.*" Some of the BIT courses are chiefly theoretical, some are chiefly practical, and some blend theory and practice. In the BIT curriculum, theoretically oriented and practically oriented courses alternate. Theoretically oriented courses prepare students by giving them conceptual and theoretical understanding of computing, and in practically oriented courses students can take that understanding into functional use.

[1] *Notes on staff meeting*, Tuesday, June 24, 2008, 14:00–16:30

In practical courses students are required to reflect on the relationship between what they have learned in their classes and on their practical work. Practical courses are not, however, intended to be the end of cycle, but they are intended to provide students with motivation and orientation for the next courses.

3.3 Problem-Based Orientation

One of the biggest open questions with problem-based learning is the choice of problems. When one thinks about problems for classroom, one needs to distinguish a problem from a trivial question to which the answer is known without any need for reflection [10] ("What color are the pants I'm wearing?"). Similar, a problem must be distinguished from *a task*: although some tasks can indeed be problematic, not all tasks are problems and not all problems are tasks. Another common misuse of the term is using it in a situation where problem is associated simply with not knowing [10]. For instance, not knowing how many provinces there are in Tanzania is not really a problem for most people most of the time.

It has been suggested that one valid use of the term *problem* is as follows: "If an obstacle occurs in the course of someone's own existence, and he/she does not know how to overcome the obstacle, then he/she has a problem." [10] A problem has two sides—the subjective side that is the feeling of necessity, and the objective side that is the situation that puzzles the consciousness [10]. In other words, in each authentic problem there is an obstacle (objective aspect) that a person wants to overcome (subjective aspect). In the BIT program's problem-based pedagogy the subjective aspect—the personal need to overcome the obstacle—is emphasized.

In addition, it must be understood that in the field of information technology problem solving (in the sense of instructions or general rules on how to overcome a problem) depicts only one aspect of the more general concept—problem management [32]. A solution—whether it is a definite unequivocal answer to a problem or, e.g., a resolution rising from a debate—is only one stage in that process [32]. Problem management may also involve identifying, comprehending, specifying, expressing, formulating, solving, and evaluating the problem in question [32].

We wished to refrain from using pseudo-problems (manufactured problems, imposed problems) in the classroom; instead, we wanted students to get acquainted with typical problems in an authentic setting. That approach, we believed, would avoid learning the problem management principle of what-you-*know*-is-what-you-get rather than what-you-*need*-is-what-you-get. The former principle builds on a specific set of features, and it responds to closed and well-constrained problems [32]. The latter principle states a "customer's problem" (sometimes in an obscure way), but it gives more room for innovativeness. When problems of the first kind are introduced at the school, a teacher often expects the student to come up with not any solution but with a solution that the teacher is familiar with [1, p.9]. Contrary to that, in the BIT program students are exposed from early on to the kinds of problems they will be working with in their work life. That, however, turned out to be much harder than it was first thought to be.

First of all, it is hard to create links between local businesses and an academic program. Companies have been very reluctant to host Tumaini's students for either long-term internships or short visits. Some of the students had to find internship places very far from Iringa, and despite numerous attempts, some were unable to find an internship place altogether [2].

Second, teachers in both hardware-focused and software-focused courses in the curriculum found problem-based orientation to be challenging. On the hardware side, organizational and administrative matters hindered the possibilities of working on real-life problems. Firstly, due to the high risk of information leakage we cannot allow students to enter staff members' offices or to work with staff members' computers. In Tanzania, leaking out exams or student information is a constant trouble and the administration does not want to risk any leaks. Secondly, the university has grown rapidly in terms of number of students, and facilities are scarce. Therefore, it has been difficult to allocate a suitable workshop space. Currently we have a 20-foot container for hardware workshop, which is not very well suited for the purpose. Thirdly, problem-based learning requires much more time from teachers than more traditional pedagogical approaches do. Currently, staff members have a heavy teaching load (e.g., teaching the basics of IT for students in all faculties), they have a large number of technical duties (e.g., IT support, installation, and maintenance), and they have a number of other duties (e.g., departmental duties, administrative tasks, research, and continuing education)[3].

After starting to work with an NGO (non-governmental organization) *Global Outreach*, which focuses on providing Internet libraries to Iringa region[4], the problem-based learning approach turned on a new gear. In May 26, BIT students assisted in setting up one 12-computer Internet center, and there is an agreement on employing Tumaini's IT students for maintenance and support work in the NGO's nine Internet centers in the region. Each month students complete a maintenance checklist in one Internet center (18 tasks ranging from maintaining peripherals to virus checks and updates), they will verify the computer inventory, they will review the findings with each school's headmaster, and report all activities to the NGO[5].

3.4 Context-Sensitivity

The very concept of *context* is multidimensional and excessively ambiguous. Even when one agrees with the importance of contextual understanding, taking context into account in pedagogy and curriculum design is neither easy nor straightforward. For example, we discovered, the hard way, in the BIT curriculum a clash between I) the aim of providing hands-on learning experiences, II) the original curriculum design, and III) students' background. Java programming classes were planned to begin in the first semester, and programming was planned to be taught in a hands-on manner[6]. The original idea was that programming should be taught using Jeliot 3, which is a program animation tool that

[2] *Internship Reports, 2008*
[3] *End of Semester Report*, Monday, September 1, 2008
[4] *Memo: Collaboration Between Global Outreach and Tumaini's B.IT Program*, Tuesday, May 13, 2008
[5] *Appendix 1: List of Tasks at Global Outreach Sites*, Tuesday, May 13, 2008
[6] *Bachelor of Science in Information Technology Degree: Syllabus*, Version 17, pp.6–7

is aimed at helping students to understand object-oriented programming [29].

The idea for the hands-on programming course was that students would surf the course material using a standard web browser, would copy-paste or type some code examples to an editor window, would compile and execute the source code to experiment on the program, and would save their files for later use. However, 89% of Tumaini's IT students had never used a computer of any kind before coming to the university. When the programming course began, students did not know about surfing, browsers, copying, pasting, typing, code, editors, windows, compiling, executing, saving, programs, or files (or about any other computing concepts for that matter.) As the programming courses plowed through, it became crystal clear that either programming courses should be postponed later in the curriculum or the idea of hands-on, practical learning of programming should be abandoned. We chose the former option and restructured the curriculum so that programming courses begin in the second semester of studies.

No-one ever clarified to BIT program's staff members what contextualization implies for their own teaching. Also students were puzzled by the idea of contextualized curriculum. One student wrote, "*I have been hearing every now and then that IT course here at Tumaini differs from other Universities curriculum adopted. That is "Contextualized." The fun thing is that, I have not noticed exactly the meaning of the term "Contextualized teaching and curriculum". Are we really being taught under that concept?*"[7] Indeed, contextualization is a multifaceted idea and can appear at several levels [37]. From our point of view, firstly, contextualization entails the idea that an IT educational program must take into account students' previous knowledge. Secondly, it entails the idea that a curriculum must consist of topics that are relevant to the geographical, technological, and socioeconomic environment where the graduates are going to work. Thirdly, it entails the idea that culture and society cannot be considered to be external to an educational program.

In the BIT program, the first idea above is gradually being better understood, and curriculum is being reshaped to better accommodate to students' level of technological literacy and educational background[8]. The first idea must, however, be better addressed through pedagogy, and we are currently conducting design research on the topic. The second idea is somewhat addressed in the curriculum, and research is currently being conducted to explore the social, economic, industrial, cultural, and all other kinds of aspects of relevance regarding the curriculum. The third idea is addressed especially in classroom teaching and problem-setting.

Currently Tumaini's BIT program is somewhat different to other university-level computing programs in terms of local and contextual concerns, but we are not certain if the program is locally relevant enough to call it 'contextualized.' But there again, there is no agreement about how much of the curriculum should be 'universal' and how much should be 'local.' Excessive contextualization may undermine another principle of the program: international recognition. That is, after their graduation students must be able to work effectively in local industry and be able to explore, identify, appreciate, and solve local problems—but they also must be able to continue their education to master's level in other institutions. The program must be balanced; it should have some local and some global elements.

[7] *Personal email*, Wednesday, July 2, 2008 (verbatim)
[8] *Curriculum Review*, June 28, 2008

3.5 Interdisciplinarity

The paradox of specialization in Tanzania is that there is such a lack of specialists in many crucial branches of technology, that Tanzanian IT professionals cannot afford specialization. That is, one cannot assume that there is readily a specialist for all the auxiliary problems that an IT professional faces. Usually one cannot hoist to others some parts of the problem knot, or the job will not get done. If an IT installation has a problem with electricity, it may take a long time to get a qualified electrician to fix the problem; if there is a problem with poor standards of building and wiring, it may be hard to find a building contractor or electrical engineer to analyze and overcome the problem. The necessary technical skills for Tanzanian computing professionals include aspects of, for instance, electrical engineering, material science, architectural design, and carpentry. Tanzanian IT professionals must be able to work with a wider range of issues than their Western counterparts.

A combination of computing fields (computer science, information technology, information systems, and computer engineering) has in general had a more positive effect on the BIT program than a negative one. In the academic year 2008–2009 the program staff includes people from computer science (B.Sc, M.Sc, PhD), computer engineering (M.Eng), and information technology (B.Tech / polytechnic). This variety provides students insight that a narrowly focused, specialized program could not provide. Technically oriented courses, theoretically oriented courses, and engineering-oriented courses can, in the BIT program, all be taught by people who specialize in those branches.

Students, however, have not been fully satisfied with the wide variety of IT skills in the curriculum. The first year students have been puzzled by the variety of IT fields and have wondered their difference. In a sense, their puzzlement is warranted: the aims of B.Sc, B.Eng, and B.Tech degrees are indeed different from each other. It has been a source of constant debate whether the BIT program should focus on teaching the latest tools and techniques, the processes of building those tools, or some deeper and more timeless principles behind those tools. Students also hold the idea that when they graduate and are hired to a company, they should be ready to start productive work from day one. But the IT job market in Tanzania is almost as wide as the job market in industrialized countries, and a general university education cannot prepare one to be expert in all IT fields. Therefore, we have planned a series of classes in career development: searching for work, applying to a job, writing a CV, preparing for a job interview, learning continuous education, and accommodating to changes of career.

3.6 International Recognition

In the BIT curriculum plan and program description international recognition was mentioned as a principle, but how to achieve that recognition was left open. We chose a two-fold approach to achieving recognition: firstly, we aim to show clearly how the program connects with the international CS, CE, and IT curricula [2, 3, 4], and secondly, we aim to demonstrate the contributions of deep local understanding to computing knowledge in general. We approach the former aim by requiring teachers to connect the topics taught in each class with core knowledge topics found in ACM/IEEE curricula. The latter aim is a research topic currently undertaken by several researchers from Tumaini's staff and collaborating institutions.

3.7 Basis on Research

A significant amount of research was done for developing this program. One doctoral thesis dealt solely on the development of contextualized IT education at Tumaini [40]. Several journal articles and conference papers were published on the development steps involved in the process [8, 19, 26, 33, 41, 42]. The foundations of culturally meaningful understanding of computing were elucidated in several articles [36, 37, 38].

Ongoing research-based development of the program is well underway: A thick description, analysis, and understanding of the issues connected with contextualized IT education is being woven from interconnected research strands. After starting the program, a number of visiting researchers have undertaken research on the program and in the program. A Spanish researcher working in Finland conducted research for his doctoral thesis on a program animation tool in the BIT program's first programming course [28]. Two Danish students did their M.Sc thesis research on social empowerment through the BIT program [24]. A Tanzanian student evaluated, in his M.Eng thesis, the contextual aspects of the BIT program [25]. A Kenyan student compared, in her M.Sc thesis, the standards of contextual sensitivity in five African IT programs, including Tumaini [13]. In addition, there are several ongoing research processes (including one ethnographic two-year research), several small interconnected pieces of research (e.g. [35]), and doctoral research work.

3.8 Organizational Matters

Although we have had to learn many lessons the hard way, we got some things right from the beginning. The most important has been *a long-term relationship* between stakeholders. Organizational knowledge and understanding can only be gained through long and active relationship, and through working with and within the organization for a long time. Trust cannot be established overnight or over the Internet. Before starting the program there already was a ten-year history between the college and the partners of the college, such as University of Joensuu, North West University, and University of Southern Denmark. In addition, a number of people and organizations, such as the Finnish Evangelic Lutheran Mission (FELM) and Tumaini's previous ICT directors hired by FELM, had successfully established trust between international networks and the university. This long-term relationship paved the way to successful tightening of the collaboration.

Second, *thorough understanding of the university organization*, culture, bureaucracy, and politics is indispensable to success. In the establishment phase of the BIT program key people—the provost and top administration, as well as the ICT director—had a thorough understanding of how decision-making in the university works. In the implementation phase, the ICT director was well positioned in the organization and had good knowledge of the workings of the university organization. Without this organizational understanding, amendments to and changes in the curriculum and program execution would have been very slow and many administrative issues would have been frustrating. Our experience underlines the quintessence of local, Tanzanian managerial knowledge as well as international collaboration.

Third, a *strong support of the university administration* is fundamental to success. From the very beginning, the provost and deputy provost for academic affairs were strong supporters of the program, and the deputy provost for administration looked favorably at the necessary (quite remarkable) funding requests for the program. Throughout the program implementation, the support of top administration has been imperative to success. Our colleagues working with similar program plans in another college in East Africa report that their work is nigh impossible due to the adverse attitude of the administration to computers and ICT in general [9].

Organizational Structure

Earlier in this paper we noted that BIT program did not fit well under any of Tumaini's four faculties, so in the beginning the BIT program was not associated with any faculty. In the beginning, this turned out to be a great asset for the program. Although faculty provides a clear position and weight within university organization, it also entails rigidity and bureaucracy. Departments at Tumaini have to get decisions approved on three levels: on departmental level, on faculty level, and on administrative or academic board level. Some decisions require an additional decision on university senate level. Not being associated with any faculty exempted the BIT program from faculty-level debates and freed it from intra-faculty competition.

Problems began to arise soon, though. Firstly, about halfway the academic year students noticed that not belonging to any faculty means that students do not have a representative in the university's academic board. Secondly, the national student loans' board requires that every student must belong to some faculty. Thirdly, special 'faculty requirements' determine how much money a student is eligible to get from the loan board. Fourthly, official forms have a box for a faculty stamp. These might not be issues in many other countries, but Tanzania's governmental institutions are notorious for the inflexibility when it comes to paperwork. Already in the very beginning of the program implementation it became clear that none of the existing faculties were willing to consider IT to belong under their compartment, so it was necessary to find an alternative organizational arrangement.

Because of these issues, in an administrative meeting in the middle of the second semester it was decided that an ICT Directorate will be founded, and that the BIT program will belong to that directorate[10]. The directorate answers directly to the university management—regarding academic issues to the Deputy Provost for Academic Affairs (DPAA), regarding administrative and financial issues to the Deputy Provost for Administration (DPA) and generally to the Provost. This arrangement, which is not uncommon in Tanzania, suits well a situation in which faculty is too large an organizational unit, but in which there is still a need for a unit that has the status and functions similar to those of a faculty.

[9] *Personal SMS*, June 22, 2008, 18:04

[10] *Internal Memo: BIT Program Planning Meeting*, Friday, April 4 2008, 15:00–16:00, §12

3.9 Skills and Competences

Initially, there was great ambiguity about what exactly are the competences that graduates of the program would be prepared for. It was unclear if those competences are mainly about theoretical competence (computer science), processual knowledge (computer and software engineering), or application knowledge (technological disciplines). This ambiguity was reflected in students' comments and questions about the program as well as in staff members' emphases in their courses. Following a workshop on skills development, organized by the college, five competence areas and a number of essential skills were explicated and included in BIT program description[11].

Five Competence Areas

A number of authors have described the key competences that IT professionals must have (e.g., [15, 30]), and those competences are presented, for instance, in SIGITE Curriculum Committee's IT2005 model curriculum. The competences listed in texts like the ones above are numerous and very general, but for the BIT curriculum we selected and outlined five areas of competency: network administration, web development, hardware support, programming, and academic competency.

Network administration competency. One of the most urgent skills needed in the ICT sector in Tanzania is network administration. In the BIT program, competency in network administration is built through six regular courses, and those who wish to go deeper into the topic can support the regular courses with one to three application courses and elective courses, as well as with two internships. The competence track begins with introductory, theoretically oriented courses, which also teach students some practical aspects, such as punching network cables and building cable runs. The courses continue to networking concepts in operating systems, building blocks of networks, network services and protocols, network security, and network troubleshooting.

Web developer competency. The dotcom boom never reached most parts of Tanzania. Most of Tanzania's enterprises, organizations, and institutions do not have visibility in the Internet. Electronic services and operations, such as e-Learning, e-Business, distance work, and e-Government are largely unknown. Developing these operations and developing services for yet-underdeveloped information society requires an army of competent web developers.

Hardware support competency. Perhaps the single most important competency for Tanzanian IT professionals is competency in hardware support. For instance, many computing facilities are seriously underutilized due to hardware failures that could be repaired relatively cheaply and easily[12]. One of the main reasons for the large number of hardware problems is that a large percentage of Tanzania's computers are donated from Europe or the U.S., and the donated, used computers face hardware problems much more often than the brand new computers in industrialized countries do. Many of those hardware problems could, however, be easily fixed with just basic knowledge of computer hardware and with some improvising. Also, knowledge about the most common causes of hardware problems (dust, UV radiation, humidity, heat, and vibration [11]) is essential for avoiding hardware problems in the first place.

Programming competency. Programming is an essential skill for any computing professional [4]. One could argue that programming is not a *central* part of an IT curriculum, because programming is more of a productivity and development activity, whereas IT is a field focused more towards analysis, deployment, installation, and maintenance of IT systems (cf., e.g., [9]). However, many of the tasks of IT professionals require programming knowledge. In addition, modern web development tasks require programming and database management skills.

Academic competency. As described earlier, one of the key principles in Tumaini's BIT program is that it ought to be internationally recognized. That is, graduates of that program should be able to pursue further studies in internationally recognized Master's level programs worldwide. In their further studies, B.Sc degree holders must be able to produce academic text and to understand, critically analyze, and combine academic readings.

Skills

In the course of time we have come to learn that Tumaini's IT curriculum also teaches (or should teach) a number of skills that are not explicitly mentioned in the curriculum, but which should be explicit. Below we introduce ten skills that are a part of a Tanzanian IT professional's vocational proficiency, and which are currently being explicitly incorporated to the program content.

Core IT Skills. Naturally, the most important skills for an IT professional are skills related to developing, analyzing, deploying, implementing, constructing, maintaining, installing, and troubleshooting information technology equipment. The core skills of a Tanzanian IT professional are similar to those that ACM/IEEE IT Curriculum [4] addresses. (Note, however, that in addition to the five core competences we described above, a Tanzanian IT professional scarcely copes without, for instance, some electrical engineering and some material science knowledge (e.g., [22]).)

Team-working and networking skills. Work in computing industry is usually done in teams. In addition, many IT professionals are free agents, who work on a project basis, and whose income depends on their networking skills. In BIT program, team-work is practiced not only in the classroom, but also through the many practical assignments and internships. For instance, when students work at the nine Internet centers of the NGO Global Outreach, they need to work with the lab managers at each center and with headmasters of the schools where the centers are located.

Interdisciplinary and innovation skills. Information technology is a service operation, so IT professionals need to be open to the problem management styles and requirements of other academic fields, industries, and society. Especially in developing countries, where material conditions and availability of tools may be limited, and where solutions and innovations must often be creative and innovative, IT professionals need to transcend boundaries of academic fields, cultures, and professions.

Leadership and management skills. In Tanzania there is a dire shortage of skilled and educated IT workforce, and a Bachelor's degree is a relatively high degree. Many students will assume managerial positions in their organizations, and they must be prepared for leadership. Practice

[11] *Notes on Workshop on Skills Development*, Tuesday, May 6 2008, 9:00–16:00

[12] *Field Notes: Pomerini Secondary School*, October 13, 2008

with team dynamics and managerial skills is incorporated in the curriculum through assignment of roles in teamwork. In addition, students can take elective courses in management topics.

Law, ethics, moral, and work ethic. Awareness and knowledge of the social aspects of information technology is essential for IT professional. Already in the first year, students get acquainted with the ethical codes of ACM and IEEE as well as with typical cases in computing ethics. In the second year of the curriculum the Faculty of Law organizes for BIT students a course on Tanzania's cyber laws. But we consider ethical and moral issues to be of such importance that those issues must be woven to several courses in the curriculum (cf. [7]).

Sustainable planning skills. IT systems are a major investment for Tanzanian individuals and organizations. The investment must be useful over a long period of time, it must be usable by various people without difficult training, and it must be maintainable by various people. Any IT system must be planned to serve users over an extended useful life span. Hence, life-cycle analysis and design for prolonged operation are topics in several courses.

Life-long learning skills. The terrain of IT is constantly changing, and every fiscal quarter seems to bring along new things that one should know or be able to do. Life-long learning is a skill and an attitude that students must assume in order to continue to be professionals throughout their career. We encourage life-long learning as an attitude by requiring students to individually select, study, and analyze topics in a number of courses.

Customer service skills. Unfortunately, in Tanzanian office world customer service is an underrated concept. Too often those with the skills and knowledge to solve a problem use their position to gain social status, money, and political leverage instead of focusing on how to solve customer's problems efficiently. Students must understand that it is a matter of professional responsibility, ethics, and pride to *serve* customers. Students must understand that a positive attitude to customer service is a sine qua non of establishing and maintaining a good reputation, and therefore their attitude indirectly affects their income. Students practice their customer service attitude and skills through their internships as well as through practical training at Tumaini's public Internet labs.

Research skills. Although research is not a central part of a bachelor's degree, a bachelor's degree is the first stepping stone to an academic career, and bachelor-level graduates must be able to find out relevant information about topics in their field, evaluate that information, and utilize that information to inform their work. At BIT program, the B.Sc thesis project spans over two years' period and teaches students the necessary tools for their work and for continuing education on M.Sc level. In addition, critical reading skills are essential for their academic career, yet reversing the uncritical attitude that the Tanzanian educational system instills is challenging.

Globalization and social responsibility. Even though Africa does not seem to be a part of the "flat world" [17], students in developing countries must understand the forces of globalization as well as the forces that have supercharged the modern marketplace and economics. Students in developing countries bear the same responsibility over the have-nots of the world as students in industrialized do. BIT curriculum includes compulsory development studies course and other courses that teach BIT students awareness of poverty, its reasons, and some potential ways out of poverty.

4. CONCLUSIONS

In this article we have analyzed our experiences of implementing a contextualized B.Sc program in IT in Tanzania. The rich understanding that we have gained from our research and work has offered us insights and lessons that can benefit other institutions undertaking similar projects. This section pulls together the insights we have gained and the lessons we have learned.

4.1 Principles Revisited

Earlier in this paper we noted that in the course of program implementation we had to clarify the principles associated with the BIT program. In addition to clarification of the semantic content of the principles, a deeper analysis of the six program principles led us to consider those principles in two ways: firstly, the principles are not a mere bullet list of items, but the principles are interconnected; and secondly, the principles arise from two kinds of considerations or motivations.

Interconnected Principles. The program principles are interconnected and interdependent in various ways (Figure 2). First of all, international recognition of an educational program involves the idea that a non-standard program cannot be just a collection of courses, but its curricular and pedagogical aspects must be based on scientific research. Continuous research-based development of a program also sustains its international recognition. Second, sensitivity to context can come only from a thorough understanding of the context, which, again, requires continuous investment in research. Context-sensitivity also necessitates a practical orientation.

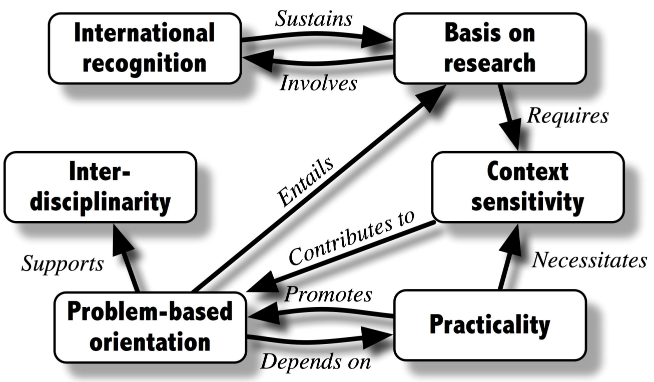


Figure 2: Interconnections of Program Principles

Third, a practical approach and a problem-based orientation are closely related. A practical approach depends on a problem-based orientation, for a practical approach requires a real-world environment and authentic problems [20, 21]. A problem-based orientation, on the other hand, promotes practical, sometimes engineering-oriented approach of getting the job done efficiently with minimal resource consumption. For the same reasons, an interdisciplinary approach supports a problem-based curriculum: a *bricoleur* in

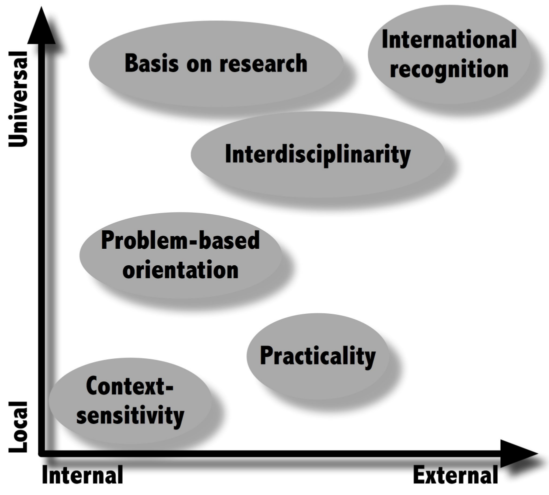


Figure 3: Local and Universal Considerations vs. Internal and External Motivations

IT fields should be able to choose whatever tools or theories he or she finds suitable for the job. Even more, the very idea of a research-based program entails a problem-based orientation. That is, research arises from unanswered questions, open problems, and lack of understanding. Especially applied research and engineering work are essentially problem-solving enterprises.

Motivations and Considerations. The six program principles arise from different kinds of motivations and considerations (Figure 3). Firstly, whereas some of the principles arise from local needs and competences, some other principles are more universal by nature. Secondly, some principles are mainly motivated by the needs of the program whereas other principles are mainly motivated by external requirements and pressures.

Figure 3 presents the program principles on two axes: a 'local vs. universal' continuum and an 'internal vs. external' continuum. The local/universal axis maps the six principles according to the locality of each principle: Concepts at the *local* end of the axis are considered to be important from a local perspective, and concepts at the *universal* end of the axis are universally accepted and adopted practices or conventions. The internal vs. external axis maps the principles according to the source of each principle: Concepts at the *internal* end of the axis arise from the program itself ("internal motivations"), and concepts at the *external* end of the axis meet requirements that are imposed by some external actors. Consider international recognition, for instance. International recognition is not important at all to the program per se (an educational program can be very successful without being recognized internationally). However, many other stakeholders—such as other academic institutions, the ministry of education, collaboration partners, and the private sector—require the program to be internationally recognized.

4.2 Lessons Learned

Roughly speaking, our analysis can be condensed to four lessons. Those lessons are based on our research material and experiences we have gained from developing and implementing a contextualized IT program. Although the lessons seem intuitively appealing, one should keep in mind that the recommendations we make are probably not generalizable to all developing countries.

Lesson 1: Ambiguities Hinder Success

In a non-standard program students, staff, and other stakeholders must have a common understanding of what the keywords regarding the program mean, how they are implemented, and why they are important. Similar, there must be an understanding of why some innovations or functions (e.g. transparency in grading) are important and how one should utilize those innovations or functions. When decisions about pedagogy, operations, communication channels, or any other aspects of the program are made, the more clarity there is about those decisions, the higher the chances for success. For instance, we did not succeed to communicate the concept of contextuality to all stakeholders, which lead to delayed adoption of the idea; on the other hand, those stakeholders who had a joint understanding of the concept of contextuality, unequivocally adopted the idea.

Lesson 2: Contextualization Is a Slow Process

In our work on contextualized IT curriculum design we did not anticipate and appreciate some crucial contextual issues in Tanzania. For instance, we did not consider the fact that most students enter university being fully computer illiterate or the fact that Tanzanian educators are mostly unfamiliar with (and even resistant to) problem-based pedagogy. Similar, although we have set up an online learning platform Moodle, although we have good online course material for some of the courses, and although the program utilizes teachers from a collaborating university to help with online teaching, most of BIT program teachers choose not to utilize Moodle. We attribute this reluctance to, firstly, the teachers' lack of experience on online teaching (in the role of a student as well as in the role of a teacher), and secondly, to the fact that online teaching demands more time than traditional instruction does [39]. It has taken us a lot of time and effort to construct a good fit between the stakeholders' strengths, program aims, organization, pedagogy, and support functions. Furthermore, this issue continues to be one of the key issues of the program's continuing development.

Lesson 3: Organization Matters

Each institution has its own organizational structure, practices, politics, channels of influence, ways and means of negotiation, administrative roles, student-staff relationship, hierarchies, bureaucracy, processes, codes of conduct, unspoken rules, and methods of working. Difficult as it may be, implementers of a successful educational program must find ways to make the organization work for the program, and not try to fight the organization. Political deftness is a rare skill, and it is difficult to find people who are able to make the system work seamlessly. Organizational experts are as important as content experts.

Lesson 4: It Is Difficult to Get Feedback

Students know best what students want. The more often they can really tell what they feel about the program, the better. In Tanzania this is, however, a difficult task to achieve. Students are not used to telling what they really think about their teachers or their institution—it seems that some students do not even believe that feedback can even lead anywhere. Others are reluctant to give critical feedback. However, when students do decide something together, they are very strongly united behind their decisions. A pitfall to this lesson in Tanzania is, however, that students often do not know enough about technology to make informed demands. A mass student movement about a poorly understood thing is scarcely constructive phenomenon and always hard to deal with.

This article is the first report of a three-year investigation of the development of Tumaini University's BIT program. In our future research, we focus on the challenges of online teaching in Tanzania, public-private partnership development, public perceptions of IT and ICT, issues of e-privacy, and course contextualization. In the end, we hope that this report encourages ambitious, alternative initiatives in IT education in developing countries, and clarifies some ambiguities regarding contextualized IT education.

5. REFERENCES

[1] R. L. Ackoff. *The Art of Problem Solving.* John Wiley & Sons, Inc., New York, NY, USA, 1978.

[2] ACM Computer Engineering Curriculum Committee. Computer engineering 2004: Curriculum guidelines for undergraduate degree programs in computer engineering.

[3] ACM Computer Science Curriculum Committee. Computing curricula 2001: Computer science, 2001.

[4] ACM Information Technology Curriculum Committee. Computing curricula: Information technology volume, 2005.

[5] M. Agar. Ethnography. In N. J. Smelser and P. B. Baltes, editors, *International Encyclopedia of the Social & Behavioral Sciences*, volume 7, pages 4857–4862. Elsevier, Oxford, UK, 2001.

[6] P. Atkinson and M. Hammersley. Ethnography and participant observation. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 248–261. SAGE, London, UK, 2nd edition, 1994.

[7] R. H. Austing, B. H. Barnes, D. T. Bonnette, G. L. Engel, and G. Stokes. Curriculum '78: Recommendations for the undergraduate program in computer science– a report of the ACM curriculum committee on computer science. *Communications of the ACM*, 22(3):147–166, 1979.

[8] N. Bangu, R. Haapakorpi, H. H. Lund, N. Myller, F. Ngumbuke, E. Sutinen, and M. Vesisenaho. Information technology degree curriculum in Tanzanian context. In P. Cunningham and M. Cunningham, editors, *IST-Africa 2007 Conference Proceedings*, volume CD-ROM, Maputo, Mozambique, May 9–May 11 2007.

[9] D. P. Bills and J. A. Biles. The role of programming in IT. In *SIGITE '05: Proceedings of the 6th Conference on Information Technology Education*, pages 43–49, Newark, NJ, USA, 2005.

[10] M. C. Borba. Ethnomathematics and education. *For the Learning of Mathematics*, 10(1):39–43, 1990.

[11] E. Brewer, M. Demmer, M. Ho, R. J. Honicky, J. Pal, M. Plauché, and S. Surana. The challenges of technology research for developing regions. *IEEE Pervasive Computing*, 5(2):15–23, 2006.

[12] F. P. Brooks, Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.

[13] L. Cheptegei. Standards and contextual sensitivity in computer science/information technology degree curricula: A case of five sub-sahara africa universities. Master's thesis, University of Joensuu, Joensuu, Finland, 2008.

[14] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[15] J. J. Ekstrom, S. Gorka, R. Kamali, E. Lawson, B. Lunt, J. Miller, and H. Reichgelt. The information technology model curriculum. *Journal of Information Technology Education*, 5:343—361, 2006.

[16] C. Ellis and A. P. Bochner. Introduction: Talking over ethnography. In C. Ellis and A. P. Bochner, editors, *Composing Ethnography: Alternative Forms of Qualitative Writing*, pages 13–48. AltaMira Press, Walnut Creek, CA, USA, 1996.

[17] T. L. Friedman. *The World is Flat: A Brief History of the Twenty-First Century.* Farrar, Straus, & Giroux, New York, NY, USA, 2005.

[18] R. W. Hamming. One man's view of computer science. *Journal of the ACM*, 16(1):3–12, 1969.

[19] C. Islas, M. Vesisenaho, M. Tedre, and E. Sutinen. Implementing information and communication technology in higher education in Tanzania. In P. Cunningham and M. Cunningham, editors, *IST-Africa 2006 Conference Proceedings*, volume CD-ROM, Pretoria, South Africa, May 3–May 5 2006.

[20] D. H. Jonassen. Instructional design model for well-structured and ill-structured problem-solving learning outcomes. *Educational Technology Research and Development*, 45(1):65–95, 1997.

[21] D. H. Jonassen. Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4):63–85, 2000.

[22] J. Kemppainen. Building ICT facilities for education in a developing country. Analysis of an ICT project at Tumaini University/Iringa University College 2000–2004. Master's thesis, University of Joensuu, Department of Computer Science and Statistics, Joensuu, Finland, December 11 2006.

[23] D. E. Knuth. Theory and practice. *Theoretical Computer Science*, 90(1991):1–15, 1991.

[24] S. Loft Rasmussen and E. Larsen. Social empowerment through ICT education: An empirical analysis of an ICT-educational program in Tanzania. Master's thesis, IT University of Copenhagen, Copenhagen, Denmark, March 3 2008.

[25] J. M. Longino. Evaluation of implementation of BSc IT curriculum at Tumaini University. Master's thesis, Lappeenranta University of Technology, Lappeenranta, Finland, September 2 2008.

[26] H. H. Lund, J. Nielsen, E. Sutinen, and M. Vesisenaho. In search of the point-of-contact: Contextualized technology refreshes ICT teaching in Tanzania. In *Proceedings of the Fifth IEEE International Conference on Advanced Learning Technologies, 2005. ICALT 2005.*, pages 983–987, July 5–July 8 2005.

[27] K. Mgaya. Development of information technology in

Tanzania. In E. P. Drew and F. G. Foster, editors, *Information Technology in Selected Countries*. United Nations University, Tokyo, Japan, 1994.

[28] A. Moreno. Program animation as a learning scaffold. Unpublished Manuscript, 2008.

[29] A. Moreno and M. S. Joy. Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science*, 178:51–59, 2007.

[30] H. Reichgelt, B. Lunt, T. Ashford, A. Phelps, E. Slazinski, and C. Willis. A comparison of baccalaureate programs in information technology with baccalaureate programs in computer science and information systems. *Journal of Information Technology Education*, 3:19–34, 2004.

[31] G. W. Ryan and H. R. Bernard. Data management and analysis methods. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 769–802. SAGE, Thousand Oaks, CA, USA, 2nd edition, 2000.

[32] E. Sutinen and J. Tarhio. Teaching to identify problems in a creative way. In *Proceedings of the FIE'01 Frontiers in Education Conference*, volume T1D, pages 8–13, Reno, NV, USA, October 10–13 2001.

[33] E. Sutinen and M. Vesisenaho. Ethnocomputing in Tanzania: Design and analysis of a contextualized ICT course. *Research and Practice in Technology Enhanced Learning*, 1(3):239–267, 2006.

[34] M. Tedre. *The Development of Computer Science: A Sociocultural Perspective*. PhD thesis, University of Joensuu, Department of Computer Science and Statistics, Joensuu, Finland, 2006.

[35] M. Tedre and B. Chachage. University students' attitudes towards e-security issues: A survey study in Tumaini University, Tanzania. In *Proceedings of the 5th International Workshop on Technology for Innovation and Education in Developing Countries (TEDC2008)*, Kampala, Uganda, July 31–August 2 2008.

[36] M. Tedre and R. Eglash. Ethnocomputing. In M. Fuller, editor, *Software Studies / A Lexicon*, pages 92–101. MIT Press, Cambridge, Mass., USA, 2008.

[37] M. Tedre, E. Sutinen, E. Kähkönen, and P. Kommers. Ethnocomputing: ICT in cultural and social context. *Communications of the ACM*, 49(1):126–130, January 2006.

[38] M. Tedre, E. Sutinen, P. Kommers, and E. Kähkönen. Appreciating the knowledge of students in computer science education in developing countries. In *Proceedings of the IEEE conference ITRE/TEDC 2003*, pages 174–178, Newark, NJ, USA, August 11–13 2003.

[39] L. A. Tomei. The impact of online teaching on faculty load: Computing the ideal class size for online courses. *Journal of Technology and Teacher Education*, 14(3):531–541, 2006.

[40] M. Vesisenaho. *Developing University-Level Introductory ICT Education in Tanzania: A Contextualized Approach*. PhD thesis, University of Joensuu, Department of Computer Science and Statistics, Joensuu, Finland, 2007.

[41] M. Vesisenaho, M. Duveskog, E. Laisser, and E. Sutinen. Designing a contextualized programming course in a Tanzanian university. In *Proceedings of the 36th Annual Frontiers in Education Conference*, pages 1–6, 2006.

[42] M. Vesisenaho, J. Kemppainen, C. Islas Sedano, M. Tedre, and E. Sutinen. Contextualizing ICT in Africa: The development of the CATI model in Tanzanian higher education. *African Journal of Information and Communication Technology*, 2(2):88–109, 2006.

[43] L. Wittgenstein. *Philosophical Investigations*. Blackwell Publishers, Oxford, UK, 2nd bilingual edition, 1958.

A Typology of CS Students' Preconditions for Learning

Maria Knobelsdorf
Freie Universität Berlin
Takustr. 9
D- 14195 Berlin
+48-30-838-75187
knobelsd@mi.fu-berlin.de

ABSTRACT

Problems that first year students encounter when majoring in Computer Science (CS) are complex and interrelated. We assume that CS majors drop the subject because, among other non-educational reasons, the teaching process and learning environment do not fit their preconditions for learning. Before meaningful educational interventions can be developed to address this issue, a profound understanding of students' learning backgrounds is needed. For this reason, we developed a biographical research approach, which allows us to analyze students' individual computing experiences retrospectively.

Students' computing experiences are individual and thus vary. However, students still share some common experiences, beliefs, and perceptions and a certain coherence or relationship should exist between them. Therefore, the objective of our research is to reconstruct typical patterns among the single characteristics of students' preconditions. For this purpose an empirically-based typology is planned. This paper presents our research design, providing a detailed description of how to develop an empirically-based typology.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*, *Literacy, Self-assessment.*

General Terms

Experimentation, Human Factors.

Keywords

Typology, CS, Computers and Society, CS Education Research, Pedagogy, Computer Biographies.

1. INTRODUCTION

In Computer Science (CS), there is a consistently high drop-out rate, especially among female CS students. A myriad of issues contribute to this problem. For example, women may not study CS because they do not perceive themselves as computer scientists or identify with CS as a subject.

The reasons for high drop-out rates in CS are complex and interrelated and some aspects of this issue relate particularly to matters of CS Education. The latter is the focus of our research project at the Institute of Computer Science, Freie Universität Berlin. We investigate how students familiarize with CS, and we assume that this is influenced and formed by, among other factors, the interaction with CS artifacts. Our objective is to elaborate on this interaction in both formal and informal settings. For this purpose, we developed a biographical research design based on the principles of qualitative social research in which we analyze students' computing experiences retrospectively [31].

Our goal is to examine if these individual experiences comprise patterns that could be identified as types and embraced in a typology of CS students' preconditions for learning. The intent is to provide a practical and useful summary of the research field, as well as a theoretical background that could be used to develop educational interventions for teaching introductory CS classes. In this paper, we present our research approach and preliminary findings. The paper consists of four parts:

1. In section 2, we examine the drop-out problem in CS and provide a rationale for our research purpose.
2. In section 3, we present our research design. It includes the theoretical background, our research instrument, and an overview of the intended typology development.
3. In section 4, the methodology of an *empirically-based typology* is presented.
4. Finally, section 5 describes the current status of our intended typology development, summarizing studies conducted to date.

The paper concludes with section 6 where we discuss areas for future research.

2. RESEARCH MOTIVATION

Problems that first year students encounter when majoring in a subject are complex and interrelated. Before initiatives are developed to motivate more students to major in CS, it is necessary to understand why so many students who were accepted into CS programs, drop the subject so quickly. The drop-out rates in CS have always been extremely high at German universities for decades. Therefore, it is reasonable to explore the reasons for the high drop-out in CS in general.

The *Freie Universität Berlin* conducted in 2006 an empirical study throughout the university and examined the reasons why students drop a subject and leave the university [34]. Even though the results of that study are not the topic of this paper, it is interesting considering the several items of the study. The first main topic was students' *preconditions for learning*, which included the family's and the student's educational background, the reason for the subject choice, information on the subject and university to be chosen, and expectations of the subject and program. Next it surveyed students' *personal conditions* like family situation, financial situation, and health. The third main topic referred to *students' experiences* with their degree program and subject, including qualification requirements, conditions at the university, and learning experiences. Finally, the last main topic focused on the *reasons for dropping* the subject: personal and family reasons, financial situation, conditions at the university, motivation, disappointment, unmet expectations, performance and failure, and changing their profession. These items show how complex the reasons for dropping out are and how they are influenced by very different reasons like financial circumstances, how the subject is perceived, or learning environment.

As CS educators, we concentrate on matters of learning in order to contribute to the avoidance of drop-out. The preconditions for learning (including education and expectations of the subject) as well as students' experiences with the subject (including conditions at the university and learning experience) are the major points of interest. Considering the education process holistically is important, and it is crucial to how we understand learning.

From a constructivist perspective, learning is a process in which students construct knowledge and understanding individually. Students actively take part in this individual process. Learning becomes not only cognitive knowledge acquisition, but it also includes and affects all aspects of a student's personality [10]. This means that interest in and perceptions of CS do not arise suddenly; they develop gradually in a process of experience and understanding. Therefore, students enter CS class not as tabula rasa, but with some already acquired knowledge, ideas, and expectations. It is important to consider students' prior experiences and to incorporate students' everyday contexts into teaching [9], [33]. From now on, we subsume under the term *preconditions for learning* all aspects of students' educational background: every aspect of student's cognition and personality that will affect the further learning: e.g. preconceptions, pre-knowledge, beliefs, expectations, motivation, and interest.

Research aimed at understanding students' interest and involvement in CS was conducted mainly with a focus on gender. It revealed that students frequently have wrong, limited, or inadequate ideas about career opportunities in CS, as well as social environment and culture [8]. Beliefs about IT jobs and careers are highly biased and restricted to the cliché of a lonely male programmer in front of a computer-screen [27]. Students' preconceptions about CS should also be considered: Many students believe CS is primarily concerned with using and administrating computers [25]. Students who are comfortable using a computer believe to be successful in CS, as well [5], [36]. Previous research on students' knowledge when they begin a CS program confirmed their different levels of pre-knowledge [13], [16], [17], [24], [32]. Hence, it is reasonable to think that students' preconditions for learning CS have a major impact on their success in studying the subject.

We assume that CS majors drop the subject because, among other non-educational reasons, the teaching process and learning environment does not fit their preconditions for learning [3]. Before meaningful educational interventions are developed to address this issue, a profound understanding of students' learning backgrounds is needed. Hence, our research questions are:

1. What preconditions for learning do CS students have before starting university studies?
2. How do these preconditions develop and influence further learning?
3. What kind of a patterns, similarities or differences among the single characteristics of students' preconditions can we reconstruct?
4. How are these preconditions related to what is expected from students in the first year of studies?

In the next section, we present the research design that focuses on these research questions.

3. RESEARCH DESIGN

Based on our research questions, the research design considers CS students' preconditions for learning. Consistent with constructivism, we intend to examine these preconditions from the students' perspectives because we want to provide a background for teaching that allows the students' individual expectations to be met. Furthermore, we intend to examine these preconditions retrospectively because we are interested in students' perspectives on a specific moment: the beginning of their university studies. This purpose involves a biographical perspective on learning.

3.1 Biographical Research

Biographical research in education considers life as a process of learning and individuals' biographies as stories of learning. A biography (as opposed to curriculum vitae) is considered a subjective construction of reminiscent moments in life, where an individual describes particular situations and learning processes that were important for him or her. These processes refer not only to a formal setting. They also include experiences, changes, and decisions a subject went through and that established his or her self-conceptualization, world-view, and habits [11].

In his research on biography and education, Marotzki concludes that the process of creation of self-conceptualization and world-view is important for the construction of biographies: "The *perspective of individual sense- and meaning-making* leads directly to the approach of modern biographical research […] An understanding of learning and education […] becomes possible only when one comes to understand processes of learning and education as specific way of interpreting oneself and the world." ([26], p. 103). A research approach that focuses on biographical learning processes must therefore consider self-creation and world-making of individuals.

Our research design is based on this biographical approach. We are not interested in the entire biography, but in the parts that are relevant to CS Education. Therefore, we concentrate on all parts of a biography referring to learning, experiencing, and understanding CS. In particular, we are interested in every kind of interaction between one or more persons and CS artifacts. CS artifacts include both physical occurrences/values that can be referred to with the general term "information technology" as well as all non-physical occurrences/values that are referred to with the term "information science", e.g., algorithms, software, diagrams, etc. Since the students' interactions with CS artifacts comprise a broad field, our research approach focuses on the interaction with computers only. For more information, especially a detailed analysis about the role of computing experiences, see [31].

3.2 Methodology

We have developed a biographical research approach, which allows us to analyze students' individual computing[2] experiences retrospectively. Our data gathering method provides an autobiographical essay (usually hand-written) on computing experiences, which we call *a computer biography* [31]. We ask students to write down their computer biographies and encourage them to start with the first contact with a computer they can recall. We stimulate this writing process with "lure texts", which are quotes from other computer biographies. The question is intentionally open-ended to encourage the individuals to make their own decisions about which experiences were most significant. It is important to note that students are not asked about any specific aspects explicitly. The fact that certain references to different aspects occur indicates how important such experiences are to students' relationships with CS artifacts.

Computer biographies of CS majors explain why and how students chose to study CS. Such texts usually follow a typical narrative pattern and are constructed in a very coherent way. Additionally, we find important experiences that fostered or constrained the students' development. Since computing and CS are closely related (especially for novices), computer biographies reveal information about students' understanding and beliefs of CS [21].

According to the biographical perspective and constructivism, every student constructs knowledge individually and has different perceptions and beliefs about CS. Consequently, we should reconstruct the biographical learning process of each student and develop personalized interventions. However, our institution's structure and capacity make it impossible to achieve this degree of personalization. Therefore, effects of educational interventions are likely to be limited to these students, whose biographical learning processes "match" these interventions. However, interventions should reach all students.

Students' computer biographies are individual and thus vary. However, students still share some common experiences, beliefs, and perceptions. In addition, certain relationships should exist between several experiences, beliefs, and perceptions in a student's computer biography. This requires the reconstruction of some typical pathways in computer biographies and the development of a typology of students' biographical learning processes of CS.

What exactly is a typology? Typologies play a major role in conceptualizing complex social realities. A certain social reality is surveyed and empirical data is collected. A typology is the result of a data grouping process that provides a structured and reliable overview of this social reality. Data elements that correspond to one or several characteristics are merged together into one type. Types are constructed to structure and understand these characteristics with regards to their differences and similarities. This can be done with a theoretical or empirical purpose. "The construction of classes, categories, or types is a necessary aspect of the process of inquiry by means of which we reduce the complex to the simple, the unique to the general, and the occurrent to the recurrent." ([30], p. 3).

[2] The term computing refers to all kinds of computer usage and interaction.

Types and typologies can be determined by many different characteristics and for different purposes. "[Typologies] can be used for classificatory or descriptive purposes, as heuristic devices and as methodological conveniences." ([30], p. 8). Therefore, the objective of a typology is two-fold. The first purpose is descriptive and helps to structure the collected data in order to make it manageable and to provide an overview. It is convenient and useful when the social reality is extensive and of a complexity that can be reduced with a typology. The second purpose is heuristic and has a theory-building function: It is assumed that the correlation between the elements of a type is not incidental. It is reasonable that a certain relationship exists between the elements of a type . The output is of hypothetical quality and serves as a background for theory building (Kluge, pp. 43). "This capability is built into [types], since as composites they are given a structure with functional consequences, and hence types are systems." ([30], p.8)

The results of our research should reduce the multitude of elements in our computer biographies to a few groups and therefore provide an arrangement (primarily descriptive) and structuring of our research field (CS students' preconditions for learning). This will produce manageable results that can easily be used in CS class for diagnostic reasons. Our results should also serve heuristic purposes. Because our results will form a certain relationship between the elements of our research field, it will provide a theoretical background for proposing hypotheses and theory-construction in the field.

The next section presents a detailed methodology description of how to develop a typology. We rely on this methodological background in section 5, where we present the results of our previously conducted studies to serve as a background for the indented typology.

4. AN EMPIRICALLY-BASED TYPOLOGY

This section summarizes qualitative social research about the development of an empirically-based typology. Kluge[3] reviewed the main core of social literature and research about typology theories and methodology. She gives an account of this review in [18], referring, among others, to [2], [6], [7], [12], [14], [22], [23], [37]. The author was engaged in theoretical work as well as empirical research (*Sonderforschungsbereich "Statuspassagen und Risikolagen im Lebensverlauf"* of the University Bremen[4]), where she contributed as a qualitative social researcher in the domain of methodology. Drawing from her experiences, she proposes a normative model that summarizes the essential aspects of different typification[5] approaches by [12], [22], and [23]. These authors focus on different data gathering or analysis methods and generate different sorts of types. Their different proceedings can be summarized in a general model that can be adapted. Since each of these proceedings contains methods that are useful for our approach, we intent to implement Kluge's model and adapt methodology proposed by her in each stage.

[3] In this paper we refer to Kluge's work in [18]. Because the book is in German, content only is reflected. Kluge provides a brief English summary of her work in [19].

[4] Sfb 186, funded since 1988 by the DFG (German National Research Foundation).

[5] The word typification means the process of developing a typology.

4.1 Types and Typology

A type consists of a set of characteristics that are interrelated and logically connected in regards to content. Each characteristic has different parameter-values and can be understood as a dimension of comparison. Each case[6] is classified according to its parameter-values, and then the groups are compared. A type can be understood as some multidimensional space of parameter-values and is coined as a property-space. Barton & Lazarsfeld developed and described the theoretical background of property-spaces as well as multidimensional tables that represent property-spaces [4]. Table 1 is an example of a two-dimensional property-space containing characteristics A and B that are defined by parameter-values: A1, A2, B1, and B2. The numbers in the table show the arrangement of the cases in accordance to the characteristics' values (e.g., the number 10 indicates that there are 10 cases that express A1 and B1).

Table 1. An example of a two-dimensional table.

Characteristic A	Characteristic B	
	value B1	value B2
value A1	**10**	**3**
value A2	**7**	**1**

Types are developed from the grouping process. Therefore, each type should be homogenous inside (internal homogeneity) in order to form common characteristics. Among themselves, types should be highly heterogeneous (external heterogeneity) in order to broaden diversity of the research field. However, different types can form a typology only when they refer to the same property-space ([18], p. 42).

Typologies play a major role in conceptualizing complex social realities since classifications used in sciences are not appropriate for this purpose. There is a difference between a typology and a classification. A classification must be mutually exclusive and exhaustive. A type, on the other hand, combines characteristics that are not uniquely and exclusively allocated to it. There is no clear separation between types. Therefore, it is important to remember that a typology cannot reproduce the reality. Types are based on predefined characteristics and represent only a part of reality. Hence, generalizations must be handled cautiously ([18], p. 25).

"[T]he research practice is confronted with the problem how these types can be constructed systematically and transparently. In current sociological literature, there exist only few approaches in which the process of type construction is explicated and systematized in a detailed way. […] Also different concepts of type are used (e.g. ideal types, empirical types, structure types, prototypes etc.) or the concept of type is not defined explicitly at all." [19]. Therefore, Kluge proposes a four-stage model of an empirically-based typification ([18], pp. 260), which is presented in the next section.

[6] The term *case* means a data item, unit, or entity which can be a complete interview or a part of it, e.g. a certain decision every interviewed person is talking about. In our research approach a case is a complete computer biography.

4.2 A Four-stage Model of an Empirically-based Typification

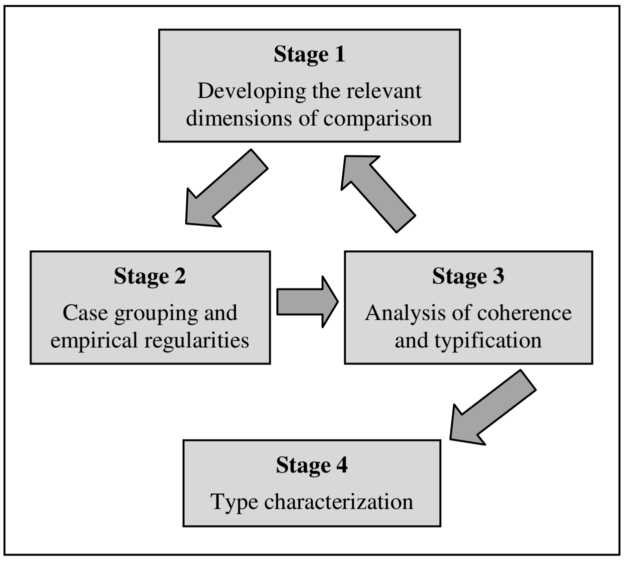
The model generalized by Kluge consists of four main stages, where the first three stages can be repeated (see Figure 1). These stages are:

1. Developing the relevant dimensions of comparison
2. Case grouping and empirical regularities analysis
3. Analysis of coherence and typification
4. Types characterization

The four stages will be described in more detail in the next four subsections.

4.2.1 Developing the Relevant Dimensions of Comparison

The first stage forms characteristics and establishes dimensions of comparison. It is important to note that each case consists of all defined characteristics. Otherwise, cases cannot be compared with each other. A typology makes sense only when all types are related to each other. Thus, this stage is very important. Only the established dimensions of comparison form the basis of typology.



Stage 1
Developing the relevant dimensions of comparison
Stage 2
Case grouping and empirical regularities
Stage 3
Analysis of coherence and typification
Stage 4
Type characterization

Figure 1. Model of the empirically-based typification ([18], p. 261).

In order to substantiate the dimensions of comparison and to form further characteristics, collected data is analyzed intensively: each case is evaluated separately and then compared to all the others. Thematic coding by Glaser, Strauss, and Corbin is frequently used. First, the data is coded with thematic keywords and then, based on the keywords, the cases are compared to each other. This way, both a case study and comparison can be combined together very effectively. Similarities and differences between the cases can be elaborated on ([18], p. 266-269).

4.2.2 Case Grouping and Empirical Regularities Analysis

After establishing dimensions of comparison and their parameter-values, all cases can be grouped. Basically, there are two ways to proceed at this point. In a bottom-up process, the two most similar cases (i.e., two cases which have the same or similar parameter value for one characteristic) are merged iteratively together into a group or cluster (agglomerative process). In a top-down process, all cases are treated as one group that is divided into sub-groups with the same or similar parameter value according to one characteristic (divisive process) ([18], p. 270).

The agglomerative process is very time consuming because all cases must be compared to each other during each step. Hence, this process is conducted with computers, and agglomerative algorithms that perform cluster analysis are used. The disadvantage of this process is that it is difficult to trace which characteristics form the cluster, and one or two irrelevant characteristics can significantly distort the result. Combinations of characteristics that do not appear in the data are not incorporated. Only digressive cases, which could not be allocated to any cluster, can be found with adequate merging algorithms ([18], pp. 275).

Multidimensional tables that represent the dimensions of comparison are helpful to illustrate the grouping process [23]. Table 2 shows an example of a two-dimensional property-space. Multidimensional tables provide "a general view over all possible combinations which are *theoretically* conceivable. Since all possible combinations often do not exist in reality and/or the differences between individual combinations of attributes are not relevant for the research question, single fields of the attribute space can be summarized." [19].

4.2.3 Analysis of Coherence and Typification

Under the presumption that characteristics do not correlate randomly, an interrelation and logical connection in regards to content between the grouped characteristics must exist. The groups or clusters that were found in stage 2 become types when this coherence and connection can be identified. This process is based on the preliminary features of each group and on further characteristics concerning similarities and differences between the cases and the groups. There is no methodological advice on how to proceed at this point. As Kluge writes, the most difficult step is to systemize the analysis of sense coherence and logical connection of the grouped characteristics ([18], p. 279).

4.2.4 Types Characterization

The typification finishes with characterizing the types as comprehensively and as precisely as possible in regards to the relevant characteristics, their combinations, and their coherence. Because the cases of one type are not entirely equal in each characteristic, the problem lies in how to picture the similarities. Different forms of types exist for this purpose: prototypes are real cases that represent the type best; ideal types present the essential characteristics in their pure form; and if only opposite types exist, extreme types are useful.

If only extreme or ideal types are used, the risk of losing diversity and the appearance of inconsistency of the investigated reality arises, since the focus lies on the pure or extreme aspects. Abbreviations of types must also be used carefully because, again, this can cause a distortion of the results ([18], p. 280).

5. A TYPOLOGY OF CS STUDENTS' PRECONDITIONS FOR LEARNING

In this section we summarize the results of our previous studies that constitute the preliminary dimensions of comparison for the typification.

5.1 The Four Dimensions of Comparison

We analyzed the computer biographies from four different perspectives: *sense*, *structure*, *habits* and *pathway*. We will summarize this approach very briefly. For further reading, see [20], [21], [31].

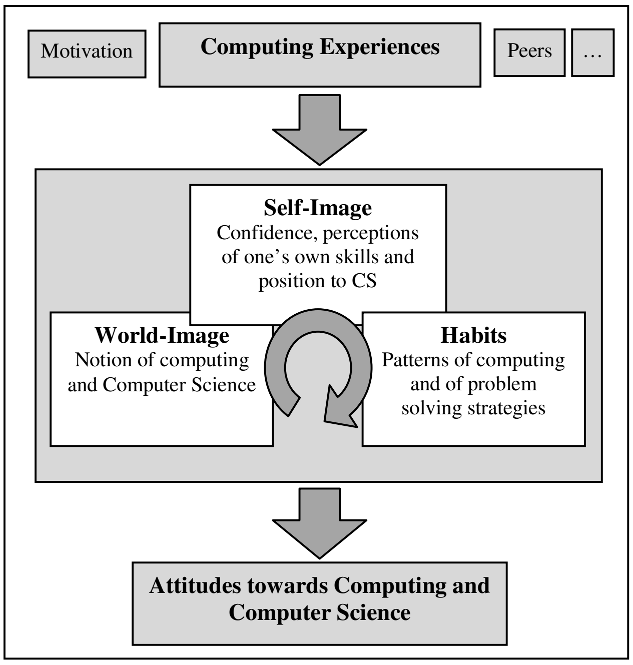

Motivation
Computing Experiences
Peers
…
Self-Image
Confidence, perceptions of one's own skills and position to CS
World-Image
Notion of computing and Computer Science
Habits
Patterns of computing and of problem solving strategies
Attitudes towards Computing and Computer Science

Figure 2. The analytical dimensions self-image, world-image, and habits specifying the biographical computing process ([31], p. 32).

Based on the biographical research approach discussed in section 3.1, we used a coding paradigm suggested by Tiefel [35] in her work on adapting Grounded Theory for the analysis of biographical learning processes: "a modified coding paradigm is proposed, with analytical perspectives geared to the reconstruction of subjective processes of making sense and constructing coherence." ([35], p. 66). However, according to constructivism, learning takes place through interaction. Therefore, Tiefel suggests considering interweaving the following perspectives into the analysis process and reconstructing the biographical learning process through them:

- The self creation of human beings and their subjective processes of sense-making are summed up under the notion of sense perspective and the self-image is then reconstructed from it. In our case, the self-image includes self-judgment and attitudes of one's own computer skills and orientation in the computer world.
- The relationship of human beings to the world and their coherence-creation are summed up under the notion of structure perspective and then the world-image is reconstructed from it. In our case, *world-image* includes personal theories and preconceptions about computing and CS.
- Finally, human beings' attitudes and contexts of interaction, reaction, and strategies of action are summed up under the notion of the habits perspective and then habits or behavior are reconstructed from it. In our case, *habits* include learning

strategies, typical performances with the computer, and reactions to problems.

These three perspectives form an analytical point of view on the holistic biographical learning process ([31], p. 31). Based on Tiefel's coding paradigm, Figure 2 illustrates an analytical approach: computing events are experienced individually and influenced by internal and external factors. These experiences are part of the biographical learning process of CS and therefore affect students' world-image, self-image, and habits related to CS. Different experiences in students' lives are interrelated. With each new experience, these three dimensions, separated only on an analytical level, are affected ([31], p. 32).

While analyzing the biographies, we realized that the biographical process is a further analytical perspective on the three dimensions because through the biographical process the world-image, self-image, and habits develop, change, and interact. The biographies of CS majors who had just entered the university revealed three periods. We call the first period the *introductory period*. It starts with the first contact with a computer. It contains experiences and situations that are initiated either by coincidence or by others. After the introductory period, a *period of development* begins. It is characterized by meaningful experiences in which students develop their interests. Then a *decision period* might take place. It contains important experiences that are crucial for the future. These experiences are described in more detail than other events in a biography [21]. Additionally, we analyzed one biography of a PhD student who graduated in CS several years ago where we examined a period that follows the decision period. Therefore, we assume that the process likely continues after the decision period.

As a result, the four dimensions (world-image, self-image, habits, and process) establish the dimensions of comparisons of our typology. These four dimensions provide many different grouping combinations, which also depend on how many dimensions each attribute has. The *process* dimension, for instance, can have three attributes: Introductory Period, Development Period, and Decision Period. Table 2 shows an example of a possible four-dimensional table of the corresponding property-space. W1, W2, S1, S2, H1, and H2 are not further specified parameter-values of the dimensions world-image, self-image, and habits and are shown just for illustrative reasons.

Table 2. Example of a possible four-dimensional table for the property-space of computer biographies.

		Process		
		Introductory Period	Development Period	Decision Period
World-image	W1			
	W2			
Self-image	S1			
	S2			
Habits	H1			
	H2			

The results of our studies (described in the next subsection) can serve as preliminary parameter-values of these four characteristics and form possible dimensions of comparison.

5.2 Parameter-Values

In our previous studies, we surveyed biographies of students majoring in different subjects: CS, Bioinformatics, Mathematics, Psychology, CS Education, and German Philology. The comparison between CS-affiliated and non-affiliated students helped in contrasting and understanding the biographies of CS majors. At the beginning, we analyzed the biographies using the Grounded Theory approach and open and axial coding ([1], pp. 271). In the last two studies, we used qualitative content analysis by Mayring [29]. We collected a large number of parameter-values of world-image, self-image, habits, and process. These characteristics are summarized below.

5.2.1 Psychology and German Philology Students

We have found the following attributes among students majoring in Psychology and German Philology: CS is perceived as a closed world that a person can only enter with special skills (a "clubhouse"). CS is an incomprehensible and complicated subject. The computer is perceived as a CS artifact and also as a tool used for working. Students' only interrelation with CS happens using the CS artifact computer.

Relating to self-image, the students believe that computer scientists are using the computer in a different way (more professional) and are able to understand "the mystery" behind it. The students believe that they are not capable of learning computer-based skills because they are missing a certain "pre-understanding" and "skills" (a special gene) that computer scientists have naturally. Therefore, these students see themselves as outsiders of the CS world.

These students are mainly autonomous learners, and they often feel helpless and left alone with problems they cannot solve and understand (learnt helplessness, attribution theory). They prefer to be taught how to use the computer and this is what they expect from a CS class at school. When using a computer, they want to understand how something works before they try to perform it themselves.

5.2.2 CS Students

We have found the following attributes among the CS majors: CS is perceived as a closed world a person can only enter with special skills, and these students think they have these skills. Based on these beliefs, the students see themselves as insiders, and the computer is omnipresent for them. As for their self-conception, they see themselves as "born to be computer scientists". They are interested in computers because they are fascinating, and computer activities are fun. They view computer problems as a challenge. They are mainly autonomous learners (learning by doing) and enjoy it. Consequently, these students often overestimate their skills and do not respond to formal learning environments.

Among the CS majors, we also found the following attributes: CS is perceived as a closed and interesting world a person can enter by changing his or her status from a user to a designer. They think that they are capable of learning things connected with a computer, and they are interested in computers because they can produce something on their own. They are mainly autonomous learners (learning by doing) and enjoy this situation, too. In contrast to the characteristics in the paragraph above, these students do not perceive themselves as being born with these skills; they accept that such skills are developed. Therefore, they are more willing to accept learning in formal settings.

5.2.3 Bioinformatics Students

Among the Bioinformatics majors, we found the following characteristics: CS is perceived as a fun and creative world, where a person can always discover and learn new things. A computer is a tool for creating. Concerning their self-conception, they think that they are capable of learning things connected with a computer, and computer activities are fun. These students are mainly autonomous learners (learning by doing) who enjoy trying things out in a playful way. In comparison to the CS students, these subjects did not identify with the computer, just as psychology students did not. But in contrast to psychology students, they were not afraid or did not feel intimidated by the computer.

5.2.4 Summary

Table 3, Table 4, Table 5, and Table 6 summarize the aforementioned characteristics according to the four dimensions: world-image, self-image, habits, and periods.

Table 3. Attributes of the world-image dimension

Attributes of the world-image dimension		
Clubhouse	**W1** CS is a closed world	**W1.1** only a person with special skills can enter
		W1.2 a person can enter by changing their status from a user to a designer
Nature CS	**W2** CS is a world	**W2.1** where a person can always discover and learn new things
		W2.2 that is fun and creative
		W2.3 that is interesting
		W2.4 incomprehensible
Nature Artifact	**W3** The computer is	**W3.1** a toy
		W3.3 a tool (to work with: a pragmatic view)
		W3.4 a tool (for creating: a creative view)

Table 4. Attributes of the self-image dimension.

Attributes of the self-image dimension		
Self-conception	**S1** Concerning the “CS world”	**S1.1** I am an insider.
		S1.2 I am an outsider.
	S2 Concerning myself	**S2.1** I was born to become a computer scientist.
		S2.2 I became a computer scientist.
		S2.3 I know that one can become a computer scientist, but this process is not completed for me yet.
		S3.2 I know that one can become a computer scientist, but I will never be one.
Learning	**S3**	**S3.1** I am able to learn things on the computer.
		S3.2 I am not capable of learning things at the computer.
Sensation	**S4** Computer activities are	**S4.1** fun
		S4.2 dull
Interest	**S5** I am interested in computers because	**S5.1** they are fascinating.
		S5.2 I can produce something on my own.
		S5.3 they are useful and helpful.
Motivation	**S6** At the computer, I’m motivated most when	**S6.1** I can do some context-based things.
		S6.2 I can perform, try different roles.
		S6.3 the activities include creativity.
		S6.4l I can work independently and be self-determined.

Table 5. Attributes of the habits dimension

Attributes of the habits dimension		
Reactions	**H1** To computer problems	**H1.1** I feel helpless.
		H1.2 I appreciate the challenge.
Learning behavior	**H2** Things I can do on the computer	**H2.1** I am a self-learner (learning by doing).
		H2.2 I was taught.
Behavior	**H3** When I do something on the computer	**H3.1** I simply try things out.
		H3.2 I try to understand things before I do them.

Table 6. Attributes of the Process dimension.

Attributes of the Process dimension		
Transition	**B1** A transition	**B1.1** has been experienced from use to design
		B1.2 has not been experienced
	B2 A development	**B2.1** has been experienced from a regular use to a professional use
		B2.2 has not been experienced
Period	**P1** Introductory Period	
	P2 Development Period	
	P3 Decision Period	

Currently, we are working on further characteristics. We examine stereotypes in CS: how students reproduce them and what kind of influence they have for successful learning [15]. We also plan a study about mindsets based on the self-theories by [10].

5.3 Further Proceedings

In this subsection, we outline how we plan to continue our research project and the intended typology. We describe the data collection, analysis, and typology stages, and we provide a timeline for these activities.

5.3.1 Data Collection

The dimensions seem to be constant. Each new aspect is a further attribute to one of the dimensions. Since all the examined attributes have been elaborated on in different studies, we were not able to compare all cases to all characteristics. These attributes form a certain dimensions of comparison but are preliminary for the development of a typology. In order to construct types, the data must be based on all attributes (see section 4.2.1). Therefore, it is necessary to survey new data that will refer to a certain dimension of comparison. It is also necessary to survey new data that will provide new attributes or further information on the existing one. For this purpose, we collected at the beginning of the winter-semester 2008 new computer biographies of first year CS students at our institute. In a second data collection step, we are planning to conduct semi-structured interviews with a subset of the same students in order to gain additional information. The intended typology will be based on this data.

5.3.2 Data Analysis

In the process of data analysis that corresponds to stage one and two of the empirically-based typology (see sections 4.2.1 and 4.2.2), we will use the qualitative content analysis by Mayring [29].

Qualitative content analysis by Mayring "[…] is defined as […] an approach of empirical, methodological controlled analysis of texts within their context of communication, following content analytical rules and step model, without rash quantification." [28]. Within this model, a category system is developed and several approaches are possible. As Mayring suggests, "[t]he main idea of the procedure is to formulate a criterion of definition, derived from theoretical background and research question, which determines the aspects of the textual material taken into account. Following this criterion the material is worked through and categories are tentative and step by step deduced. Within a feedback loop those categories are revised, eventually reduced to main categories and checked in respect to their reliability." [28].

Coding methods in Grounded Theory are not restricted, which is an advantage when a research question is open and very little is known about the research field. The disadvantage is that many steps are not standardized, nor well-defined. Therefore, a lot of expertise and capacity is necessary for decision-making and analysis. Since we have already conducted our study, we gained some knowledge and understanding of our research field. Using the typology, we aim to specify and structure our results. Therefore, we need a standardized and well-defined method to analyze our data effectively, and qualitative content analysis fits these criteria.

5.3.3 Research Schedule

Data collection is conducted in the winter-semester 2008, followed by data analysis and selection of students for interviews. The objective of the semi-structured interviews with the CS majors is to get more information on the single attributes. Next, we will collect computer biographies of the non-CS majors, analyzing and comparing them to the data of the CS majors in order to obtain a high contrast level. This data will be used for the grouping process and construction of the subsequent stages of typology. Table 7 provides an overview of the future activities.

Table 7. Overview of future activities.

Activity	Purpose
Collect new computer biographies of first year CS students (on their first day at the university) Collect computer biographies of non-CS majors Type biographies	To survey the current data To survey the contrast data To prepare for data analysis
Analyze collected data Choose students for interviews Develop semi-structured interviews	To gain some new characteristics To choose interviewees
Conduct interviews with the same first year CS students two months after they started their studies	To gain more information on characteristics
Analyze interviews (stage 1) Grouping process (stage 2) Analyze coherence and typification (stage 3) Types characterization (stage 4)	To construct a typology
Conduct interviews with the same interviewees, one year later,	To gain information about students' further learning process at the university
Analyze interviews	To examine CS program influences on further learning

In section 2, we stated four research questions we intend to answer with this research project:

1. What preconditions for learning do CS students have before starting university studies?
2. How do these preconditions develop and influence further learning?
3. What kind of a patterns, similarities or differences among the single characteristics of students' preconditions can we reconstruct?
4. How are these preconditions related to what is expected from students in the first year of studies?

Using the typology, we will answer questions 1-3 and make certain predictions about students' development in their university studies. In order to answer questions 4 we will analyze what is expected from CS students in the first year of studies and to compare this with the typology result. At this early stage, we have not developed a methodological approach for this objective, yet. Finally, we intend to conduct semi-structured interviews with the same CS students one year later and question them about their CS studies. The interview structure will be developed based on the typology.

It would be appropriate to test the typology using quantitative methods like a standardized questionnaire, but this would be an additional research project.

6. CONCLUSION

In this paper, a research design that combines different theoretical and methodological approaches from sociology, psychology, education, and CS was presented. We outlined how we adapt theory and methods to answer a research question from CS Ed (Education). How is this research design supposed to be evaluated? Certainly, it would be possible to evaluate each aspect independently. However, empirical methods from social sciences, whether qualitative or quantitative, are not “recipes” for data survey and analysis. The main challenge is that using a given methodology properly involves adaption of its epistemological and ontological context as well.

CS Ed research often approaches using methodology in an algorithmic way. However, we need empirical evidence to provide valid and sustainable results. For instance, when Grounded Theory is used to develop a theory about *student understanding of programming concepts*, the researchers have to think and behave in the tradition of qualitative empirical research, similar to sociologists. However, by doing this, they will depart from CS Ed research. This raises the question: how do we judge research design like the one presented in this paper? Overall, is such a research design still CS Ed research? Where do social sciences end and where does CS Ed starts?

The research design described in this paper intends to examine CS students’ learning backgrounds retrospectively. The intention is to analyze the preconditions for learning that CS students have and how these preconditions develop and influence further learning. Finally, the objective is to reconstruct patterns, similarities or differences among the single characteristics of students’ preconditions. For this purpose an empirically-based typology is planned. It must be discussed if this research approach is appropriate for the research purpose.

As for methodology, different stages are planned. It must be discussed if the data collection and analysis is appropriate. A clear question is whether the property-space is complete. The qualitative social research talks about data saturation, but remarks that the researchers must decide themselves when the property-space is complete. Finally, the research project presented in this paper intends to examine how CS students’ preconditions are related to what is expected from students in the first year of studies. Based on the knowledge of these preconditions, the overall goal is to better understand why CS students drop the subject due to learning reasons. Therefore, this paper concludes by asking if the intended research design is suitable to this purpose.

7. ACKNOWLEDGMENT

I would like to thank Carsten Schulte, Essi Lahtinen, Josh Tenenberg, and the reviewers for their helpful comments that improved this paper so much.

8. REFERENCES

[1] Andreas Böhm 2004 Theoretical Coding: Text Analysis in Grounded Theory. In A Companion to Qualitative Research (Flick, Uwe, Kardorff, Ernst von and Steinke, Ines, eds.). Sage Publications Ltd, 270–275

[2] Bailey, K. D. 1973 Monothetic and Polythetic Typologies and their Relation to Conceptualization, Measurement and Scaling. American Sociological Review 38, 18–33

[3] Barker, L. J., Garvin-Doxas, K. and Jackson, M. 2002 Defensive climate in the computer science classroom. In Proceedings of the 33rd SIGCSE technical symposium on Computer science education, 43-47

[4] Barton, A. H. 1955 The Concept of Property-Space in Social Sciences. In The Language of Social Sciences (Lazarsfeld, Paul F. and Rosenberg, Morris, eds.). Free Press, 40–53

[5] Beaubouef, T. 2003 Why Computer Science Students Need Language. ACM SIGCSE Bulletin 35, 4, 51–54

[6] Becker, H. S. 1968 Through Values to Social Interpretation. Essays on Social Contexts, Actions, Types, and Prospects, Greenwood Press

[7] Capecchi, V. 1968 On the Definition of Typology and Classification in Sociology. Quality and Quantity 2, 1-2, 9–30

[8] Cohoon, J. M. and Aspray, W. 2006 Women and Information Technology. Research on Underrepresentation, MIT Press

[9] Donovan, M. S. and Bransford, J. D. 2005 How students learn. History, mathematics, and science in the classroom, National Academies Press

[10] Dweck, C. 2000 Self-theories: their role in motivation, personality, and development, Psychology Press

[11] Ecarius, J. 2006 Biographieforschung und Lemen/ Biographical Research and Education[7]. In Handbuch erziehungswissenschaftliche Biographieforschung (Krüger, Heinz-Hermann and Marotzki Winfried, ed.). VS Verlag, 91–108

[12] Gerhardt, U. 1984 Typenkonstruktion bei Patientenkarrieren/Typification and Patient-careers[7]. In Biographie und soziale Wirklichkeit (Kohli, Martin and Robert, Günter, eds.). Metzlersche Verlagsbuchhandlung, 53–77

[13] Greening, T. 1998 Computer Science: Through the Eyes of Potential Students. In Proceedings of the 3rd Australasian conference on Computer Science Education, ACE 1998, 145-154

[14] Hempel, C. G. and Oppenheim, P. 1936 Der Typusbegriff im Lichte der neuen Logik, A.W. Sijt Hoff's Uitgeversmaatschappij N.V.

[15] Hewner, M. and Knobelsdorf, M. 2008 Understanding Computing Stereotypes with Self-Categorization Theory. In Proceedings of the 8th Baltic Sea Conference on Computing Education Research, Koli 2008, Finland

[7] This book/text is only available in German. The German title is translated by the author of this article.

[16] Hoffman, M. E. and Vance, D. R. 2005 Computer literacy: what students know and from whom they learned it. ACM SIGCSE Bulletin 37, 1, 356-360

[17] Kinnunen, P. and Malmi, L. 2008 CS minors in a CS1 course. In Proceeding of the fourth International Computing Education Research Workshop, ICER 2008, 79-90

[18] Kluge, S. 1999 Empirisch begründete Typenbildung/ Empirically Grounded Typification[7]. Zur Konstruktion von Typen und Typologien in der qualitativen Sozialforschung, Leske + Budrich, Opladen

[19] Kluge, S. 2000 Empirically grounded construction of types and typologies in qualitative social research. Forum: Qualitative Social Research [Online Journal] 1, 1 (2000), http://www.qualitative-research.net/fqs-texte/1-00/ 1-00kluge-e.htm

[20] Knobelsdorf, M. and Schulte, C. 2007 Das informatische Weltbild von Studierenden/Students' CS world-view[7]. In Didaktik der Informatik in Theorie und Praxis. 12. GI-Fachtagung Informatik und Schule - INFOS 2007, 69–79

[21] Knobelsdorf, M. and Schulte, C. 2008 Computer Science in Context - Pathways to Computer Science. 7th Baltic Sea Conference on Computing Education Research, Koli 2007. In Conferences in Research and Practice in Information Technology (Australian Computer Society, Inc., ed.), Sydney, Australia

[22] Kuckartz, U. 1990 Computerunterstützte Suche nach Typologien in qualitativen Interviews. In Fortschritte der Statistik-Software 2. SOFTSTAT '89. 5. Konferenz über die wissenschaftliche Anwendung von Statistik-Software. (Faulbaum, Frank, Haux, Reinhold and Jöckel, Karl-Heinz, eds.). Gustav Fischer, New York, 495–502

[23] Lazarsfeld, P. F. and Barton, A. H. 1951 Qualitative Measurement in the Social Sciences. In The Policy Sciences (Lerner, Daniel and Lasswell, Harold, eds.). Stanford University Press, 155–192

[24] Lewandowski, G., Bouvier, D. J., McCartney, R., Sanders, K. and Simon, B. 2007 Commonsense computing (episode 3): concurrency and concert tickets. In Proceedings of the third International Computing Education Research Workshop, ICER 2007, 133-144

[25] Maaß, S. a. W. H. 2006 Programmieren, Mathe und ein bisschen Hardware…Wen lockt dies Bild der Informatik?/ Programming, maths, and a bit of hardware…who is attracted by this picture of CS?[7] Informatik Spektrum 29, 1, 125–132

[26] Marotzki, W. 2004 Qualitative Biographical Research. In A companion to qualitative research (Flick, Uwe, Kardorff, Ernst von and Steinke, Ines, eds.). SAGE, 101–107

[27] Martin, C. D. 2004 Draw a computer scientist. In Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education. ITiCSE 2004, 11–12

[28] Mayring, P. 2000 Qualitative Content Analysis. Forum: Qualitative Social Research [Online Journal] 1, 2 (2000), Art. 20. http://nbn-resolving.de/urn:nbn:de:0114-fqs0002204

[29] Mayring, P. 2004 Qualitative Content Analysis. In A Companion to Qualitative Research (Flick, U. and Kardoff E. and Steinke I. von, eds.). Sage Publications Ltd, 266–269

[30] McKinney, J. C. 1969 Typification, Typologies, and Social Theory. Social Forces 48, 1, 1–12

[31] Schulte, C. and Knobelsdorf, M. 2007 Attitudes towards Computer Science - Computing Experiences as a Starting Point and Barrier to Computer Science. In Proceedings of the third International Computing Education Research Workshop. ICER 200, 27–38

[32] Simon, S., Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., Raadt, M. de, Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D. and Tutty, J. 2006 Predictors of success in a first programming course. In Proceedings of the 8th Austalian conference on Computing education, ACE '06, 189-196

[33] Sinatra, G. M. 2005 The "Warming Trend" in Conceptual Change Research: The Legacy of Paul R. Pintrich. Educational Psychologist 40, 2, 107–115

[34] Thiel, F., Blüthmann, I., Lepa, S. and Ficzko, M. 2007 Ergebnisse der Befragung der Studierenden in den Bachelorstudiengängen an der Freien Universität Berlin Sommersemester 2006. http://www.ewi-psy.fu-berlin.de/ einrichtungen/arbeitsbereiche/schulentwicklungsforschun g/forschung/bachelorbefragung.html

[35] Tiefel, S. 2005 Coding in terms of Grounded Theory. Modifying coding guidelines for the analysis of biographical learning within a theoretical framework of learning and education. ZBBS 6, 1, 65–84

[36] Turner, E. H. and Turner, R. M. 2005 Teaching entering students to think like computer scientists. In Proceedings of the 36th SIGCSE technical symposium on Computer science education, 307–311.

[37] Weber, M. 1988/1904 Die "Objektivität" sozialwissenschaftlicher und sozialpolitischer Erkenntnis/ The Objectivity of the Sociological and Social-Political Knowledge. In Gesammelte Aufsätze zur Wissenschaftslehre (Winckelmann, J., ed.). Mohr, Tübingen, 146–214

Understanding Computing Stereotypes with Self-Categorization Theory

Michael Hewner
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia, USA
hewner@gatech.edu

Maria Knobelsdorf
Freie Universität Berlin
Takustr. 9
14195 Berlin, Germany
knobelsd@mi.fu-berlin.de

ABSTRACT

The partly completed study presented in this paper explores characteristics of *stereotypes in Computer Science.* The study describes student autobiographical essays about computing, analyzed with particular attention to the ways in which students use computing stereotypes. We describe how self-categorization theory, taken from the psychology stereotype literature, might explain the essays we see and discuss potential implications of self-categorization theory on CS Education in general.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education, Literacy, Self-assessment.*

General Terms

Experimentation, Human Factors.

Keywords

Stereotypes, CS, Computers and Society, CS Education Research, Pedagogy, Computer Biographies, Categorization, Group Identity.

1. INTRODUCTION

Students in Computer Science (CS) have to cope with negative stereotypes associated with the field. Previous research has shown that stereotypes are frequently mentioned when students consider reasons not to further their CS education [7, 13] . Bigger's [3] study of CS student retention shows evidence that students leaving CS had more negative stereotypes of computing careers then those who did not. We know that some students embrace stereotypical "nerd" behaviors but that many others distance themselves from them [10]. Some advocate improving the image of CS in order to improve retention [11]. However, an examination of the literature on stereotype psychology strongly suggests that attempting to change stereotypes is problematic [12]. Before CS educators attempt to change stereotypes, we should further consider the effect of stereotypes on thinking.

In the in-progress study presented in this paper, we analyze with regard to stereotypes a set of computer autobiographies written by students. The preliminary results are examined using self-categorization theory. This psychological theory describes how stereotypes are used to create individual identities and we present it in the second part of the article. The paper concludes with a discussion section and some potential implications of this theory to CS education and teaching.

2. EMPIRICAL STUDY

Even though stereotypes had not been the subject of our previous empirical studies [5, 6], we observed that students often mentioned computing stereotypes. In the current study, we reinvestigate empirical data with the focus on stereotypes. This data consists of students' autobiographical essays about computing experiences (computer biographies). We asked students to describe experiences about computing but otherwise the question was intentionally left open-ended in order to encourage the students to make their own decisions about what experiences were most significant.

"Stereotyping is the process of ascribing characteristics to people on the basis of their group membership." [9] We used this definition as we asked the following research questions:

1. Do students mention stereotypes, explicitly or implicitly, in their biographies?
2. What kinds of stereotypes do they mention?
3. Do they use stereotypes in order to explain their decisions, behavior, preferences, or interests with regard to computing and CS?

In the current study, we examine 271 biographies: 244 biographies were written by German university students, 27 biographies were written by US college seniors. In both samples, some of the students are CS majors and some are studying majoring in non-computing fields. For more information about data collecting see [5, 6]. Though students were not asked about stereotypes explicitly, considerable references to stereotypes occur in our autobiographies. This indicates how important stereotypes are to student's relationship with computing. The next section describes our analysis of the autobiographies' use of stereotypes.

2.1 Qualitative Content Analysis

Qualitative content analysis by Mayring is a methodology from qualitative social research used to analyze systematically textual data. "The main idea of the procedure is, to formulate a criterion of definition, derived from theoretical background and research question, which determines the aspects of the textual material taken into account. Following this criterion the material is worked through and categories are tentative and step by step deduced. Within a feedback loop those categories are revised, eventually reduced to main categories and checked in respect to their reliability." [8] This procedure can be divided into five consecutive steps in which a category system is developed. These steps will be briefly explained next, illustrating the analysis we have done so far. Because of space reasons we will not describe the complete category system in detail. Instead, we focus on the categories that are relevant for our preliminary results.

In the first step of qualitative content analysis, the relevant text samples are chosen out of the complete data sample in accordance to the research questions and the theoretical background. In our study, we chose samples explicitly mentioning stereotypes (research question 1). We looked for groups ascribed characteristics or attributes and how the students positioned themselves in relation to such groups (research question 2 and 3).

In the next two steps, the relevant text samples are grouped together in accordance to principal topics that can be found in the text samples, and, from them, subtopics are generated. In the forth step, all topics and subtopics are explicated by defining categories and subcategories that describe exactly when a text sample is part of a category or not. Very often typical examples are provided with the category. The categories, together with coding rules and related textual passages, form the category system.

Based on the chosen samples we generated the first main category ***Stereotypes,*** with two subcategories: *socializing aspects* (characteristics that refer to social life: contact to other individuals and attitudes towards them, hobbies, dress and lifestyle) and *gender aspects***.** Students distanced themselves from certain aspects of computing. From these text samples, we generated the next main categories: ***Differentiation*** from (the subcategories): *stereotypes, partial knowledge, CS class in school, CS, computers*; ***Affiliation*** with (the subcategories): *CS class in school, CS, Computers*; and ***Refusal of CS/computers*** due to (the subcategories): *fear, incompetence, dependence, feelings of uselessness.*

Students identified with or distinguished themselves from groups. We denoted this in the main category ***Self-Image***. We found that individuals were describing grouping processes and used them to explain why they considered themselves interested in CS or not. We generated four subcategories: *grouping process, user, non-expert, expert.*

Once the category system is defined, 10-50% of the data is coded. After the first coding pass, the category system is revised and extended as the last and fifth step. Then the final data coding with the complete data sample is performed. The final coding will be done by more than one person in order to measure intercoder-reliability.

We have revised the categories described in the paragraphs above and currently we are now in the process of final coding using the MAXQDA coding-software [1]. Because we are still in the coding process, these results must be seen as a preliminary outcome.

2.2 Differentiation from Stereotypes

Students (especially non-computing students) frequently use stereotypes in a negative way to differentiate themselves from “nerdy” computer experts:

> *“My prejudices concerning computers, which make things more complicated instead of making them simpler, were confirmed. I felt helpless and always needed to ask my flat-mate’s boyfriend for help. He was a real ‘freak’ and was able to help quickly in most cases, but I always felt uncomfortable, due to the fact that I seemed ‘stupid’, and guilty, because I didn’t keep in touch with him otherwise (well, computer-freaks are usually boring) and I felt I was using him.* [07P1979wU6].

Biographies frequently reproduce negative stereotypes. Many of our biographies use the negative stereotype as a way of explaining their own problems with computing:

> *“I didn’t experience any further improvements with the computer, I hadn’t had a Commodore 64 like my other friends, but wasn’t interested in games only either, so I dissociated myself from the computer and I thought it sucked. In the 11th grade, I had to take CS classes, in which we used DOS. Unlike my friends, I had no clue about it. The computer became a nightmare.”* [10P1982wU6]

2.3 Affiliation with Stereotypes

Students often use stereotypes to describe their own identities. In this biography a student who previously enjoyed computers describes how he left computers for a more “punk” image:

> *“In that period I found a computer somehow un-cool and I switched to guitar. This way I got to know many musicians. We practiced hard, started bands, played gigs and were as punk as our stomachs could take it and as much as our parents allowed. I would use a computer only when necessary.”* [22ImU8]

Though his reasons for leaving computers are unclear to him, it is clear that being a punk helped him clarify a “cool” identity in his social group.

Similarly, “cool” aspects of computer culture can be attractive for students with interests in computers:

> *“Several years later I saw ‘Matrix’ in a cinema. Neo, a young hacker, was able find something out due to his computer knowledge exclusively. Something, that was inaccessible for the rest of the human race. This knowledge becomes the power and reason why he starts exploring the new world. This philosophical and, for me, revolutionary idea brought me to the idea of making peace with the computer again.* […] *It didn’t take long and I was searching after ‘hacker books’ in our local library. Suddenly everybody was talking about bank robberies and Trojans, viruses and worms. I dived into this world, which was more interesting than one could imagine.”* [07ImU8]

2.4 Ambiguous Stereotypes

Students who seemed to enjoy Computer Science nonetheless took special pains to differentiate themselves from stereotypes.

> *“*[...] *I was beginning to distance myself from people by becoming so closely involved with technology and unique expertise. To be frank I was a little afraid of being sucked into the CS major stereotype of being a pale, scruff poorly dressed student who knew little more than gaming, hacking, and which hardware on the market was the best* […]. *With this realization, I decided to pick up a certificate in information technology through the college of management.”* [547580242]

> *“I used it almost every day to play games or to check what it was able to perform, which made me a computer junkie immediately. However, I was busy not only with computers, but I also had a family and friends with whom I would regularly meet.”* [03ImU8]

This emphasis of distinctiveness from the CS stereotype was one of the most frequent commonalities between the US and German biographies. It was these sorts of biographies more than any other that led us to explore stereotypes more closely. Obviously, students in computing think about stereotypes frequently when asked about their relationship to computers. It was a point of concern for us that excitement about computing seem tied to negative stereotypes even in the minds of computing majors.

3. PSYCHOLOGY OF STEREOTYPES

It should be clear from the patterns we have highlighted in our biographies that students use stereotypes in complex ways when describing their relationship to computing. To help understand these results, the psychology stereotype literature represents a valuable resource. We introduce one theory here, and discuss its implications for our biography results and computing education in general.

Stereotypes are generalizations about groups ([12] pg. 26). When negative generalizations are applied broadly by members of a culture, they can lead to the prejudice and discrimination typically associated with the word “stereotype”. However, most modern psychology stereotype research theorizes that stereotype formation is a normal, not pathological, process of cognition. Stereotypical generalizations give us useful abstractions that help us understand social situations. Psychologists don’t agree on the exact cognitive structure of stereotypes. This paper draws on self-categorization theory, an explanation of stereotypes that we found provided some interesting insights into our biographies.

3.1 Categorization

The basic prediction of self-categorization theory is that individuals will naturally view a social context in terms of two groups: an in-group that is viewed as similar to the self and an out-group that is differentiated from the self. These categorizations change as the social context changes ([9] pg. 87). For example, A CS major in an introductory CS class might feel that he or she is a "CS major" and differentiate him or herself from the "computer enthusiasts" in the class. However, the same CS major might feel a great deal of kinship with the same enthusiasts at a party, regardless of major because the context divides more neatly into “computer /non-computer people”.

In real social situations, multiple categorizations are of course possible: individuals are divided by gender, major, dress, hobbies, etc. What makes a particular categorization salient is a combination of a variety of factors [9]:

- *Categorizations explicit in the situation*. The two opposing teams at a sporting event are likely to categorize along team lines. It's important to note that although multiple categorizations are frequently possible, sometimes categorization is so compelling there is no choice.
- *Categorizations relevant to personal goals.* If I am trying to find people to help me fix my computer problem, nerdy appearances might become salient
- *Categorizations with a large amount of meaning*. Categorizations with a large number of associations (like existing stereotypes) will be preferred to categorizations that do not help understand (e.g. hair color).
- *Categorizations dividing the social context*. If everyone falls on one side of categorization, it is not useful for understanding.
- *Categorizations establishing a positive identity*. If can I see myself in a relatively high status group, I will prefer categorizations that let me do that.

This categorization process naturally lends itself to the use of common stereotypes to make inferences about other group’s behavior. It also suggests that stereotypes are naturally used to understand the self. By categorizing oneself, the individual can incorporate a group identity into their view of themselves (at least until the social context changes and the salient categorizations are different).

3.2 Group Identity

A person’s categorization of others into groups affects that person’s behavior. One of the best known examples of this is called the “minimal group” effect in which individuals categorized into two groups based on arbitrary characteristics (like underestimating or overestimating dots) will, despite the arbitrariness of the categorization, favor their own group members when given the opportunity of the allocate resources between the groups ([12] pg. 238). Individuals also judge a statement made by a member of their own group to be closer to their own opinion, and opinions expressed by members of an out-group to be further from their own ([9] pg. 127-158).

When individuals categorize themselves as members of a particular group, their view of themselves becomes dependent on their perception of the group as a whole. If good characteristics are ascribed to the group, the individuals’ self-perception is enhanced by association. Individuals are apt to view positive information about groups they belong to more uncritically to maintain a positive self-identity.

Sometimes group members cannot view their group in a positive way. For example, if a group does poorly on an objective task or if commonly accepted wisdom makes positive comparison impossible (e.g. business students might have a high status when compared to physics majors on the basis of creativity, low status when compared on basis of intelligence). When a group’s status is low, it is considered to be under a group-directed “threat” [4]. Group members have a choice: When they feel a low amount of commitment to a group, they are likely to report that they are atypical of the group and potentially affiliate with other groups. Group members with a high amount of commitment emphasize the group’s homogeneity, may act in a more stereotypical way, and try to change the group status. Even if status improvement is not possible, high identifiers may continue to affiliate.

When individuals receive information that threatens their sense of membership in a group, it is considered to be a self-directed threat [4]. For example, if someone who considers him- or herself a Computer Scientist has difficulty understanding a class of algorithms (considering algorithmic understanding to be a characteristic of Computer Scientists), external evidence has called into question his or her group membership. Similar to a group-directed threat: group members that have a low amount of commitment are likely to distance themselves (e.g. just decide that they are a Computer Scientist who is bad at algorithms). Group members that feel a high affiliation are likely to take action to restore their perception of acceptance within the group – perhaps by studying that group of algorithms until they are clear.

This process of distancing oneself from a group is known as "individualization". We believe that this is the phenomenon we saw in some of our biographies (section 2.4) – students distancing themselves as from being classified as "normal" Computer Scientists, because of the implicit low status of the prevalent stereotype about Computer Scientists. Most of our computing majors' biographies expressed excitement about the field of CS itself. But despite this interest in the subject matter of CS, this distancing can be seen as evident individualization and of low commitment to CS.

4. DISCUSSION

What are some possible implications of self-categorization theory for understanding the effects of stereotypes on CS students? The first implication is that stereotypes significantly affect students' self-perceptions as Computer Scientists. This occurs even after the student is officially in the major and ought to be "cured" of stereotypical misconceptions. Because individuals are constantly adjusting their categorizations in view of the social context, seemingly "minor" social issues in the classroom can cause students to categorize themselves in opposition to other Computer Scientists and teachers.

This theory suggests that privileging particular attributes as definitive for CS is likely to have negative effects. When teachers focus on using a particular style of mental process [14] or elevate some students as exemplary Computer Scientists at the expense of others [2], they create a category of enrolled students who are not meeting the standards of Computer Science. If students have a high commitment, threatening their self-image as Computer Scientists can encourage them to work harder. But our biographies suggest that student's commitment to Computer Science may be low. If this is true then challenging student identity is more likely to exclude them than encourage them to work harder.

Self-categorization lends support to the view that CS hurts itself if it enforces one particular vision of the Computer Scientist in the classroom. By promoting different potential CS identities (one potential example could be different specializations in algorithms, systems, languages, etc.), students could be encouraged to categorize between a variety of different choices within CS rather than as CS/not CS. The more choice involved with an identity, and the more unique it is, the more likely it is to have strong affiliation. There are other benefits to strong commitment. When a group categorization becomes central to an individual's identity, the individual is motivated to act in ways that preserve the group's status in order to protect their own identity. When a group is having problems (for example, peers having difficulty in class), individuals who are strongly committed to a group are more likely to work with other group members to preserve their collective identity.

Finally, it can be important to recognize that the same self-categorization processes that affect our students also operate within the wider CS community. If as CS educators, we hope for our educational improvements to be adopted by the CS field at large, we should be aware that we can inadvertently threaten the identities of established members of our field by proposing to change CS in sweeping ways. Stereotype change is in many ways equivalent to a group threat because it casts into question the identities of people well-established under the traditional order. In this way, a potential innovation can lose the support of high affiliators who normally could be counted upon to devote significant time and energy to CS.

Going forward, we intend to finish this study and publish a more complete account of what we elaborated on. We also think that the predictions of self-categorization theory represent an interesting future research direction that should be explored in further detail. Clearly however, the study presented here suggests rather than verifies the claims of self-categorization theory and more stereotype specific research needs to be undertaken before we can understand how stereotypes affect students' self-identities.

5. DISCUSSION QUESTIONS

1. Are the predictions of self-categorization theory useful to us as CS educators?
2. What are the logical next steps, in terms stereotype research for Computer Science?

6. ACKNOWLEDGEMENTS

This research is supported in part by a grant from the National Science Foundation BPC Program #0634629.

7. REFERENCES

[1] *MAXQDA - The Art of Text Analysis*. Available from http://www.maxqda.com/ (2007); accessed August 31, 2008.

[2] Barker, L. J., Garvin-Doxas, K. and Jackson, M. Defensive climate in the computer science classroom. In *Proceedings of SIGCSE 2002* (Cincinnati, 2002). New York, 2002. ACM.

[3] Biggers, M., Brauer, A. and Yilmaz, T. Student perceptions of computer science: a retention study comparing graduating seniors with cs leavers. In *Proceedings of SIGCSE 2007* (Portland, 2007). New York, 2007. ACM.

[4] Ellemers, N., Spears, R. and Doosje, B. Self and Social Identity. *Annual Reviews in Psychology*, 532002), 161-186.

[5] Hewner, M. and Guzdial, M. Attitudes about Computing in Postsecondary Graduates. In *Proceedings of ICER 2008* (Sydney, Australia, 2008). New York, 2008. ACM.

[6] Knobelsdorf, M. and Schulte, C. Computer Science in Context - Pathways to Computer Science. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (Koli Calling 2007)* (Koli National Park, Finland, November, 2008). Sydney, 2008. Australian Computer Society, Inc.

[7] Margolis, J. and Fisher, A. *Unlocking the Clubhouse: Women in Computing*. MIT Press, Cambridge, Massachusetts, 2002.

[8] Mayring, P. *Qualitative Content Analysis*. Available from http://www.qualitative-research.org/fqs-texte/2-00/2-00mayring-e.htm (2000); accessed August 31, 2008.

[9] Oakes, P. J., Haslam, S. A. and Turner, J. C. *Stereotyping and Social Reality*. Blackwell, Oxford, 1994.

[10] Rasmussen, B. and Håpnes, T. Excluding Women from Technologies of the Future? A Case Study of the Culture of Computer Science. In *Sex/Machine: Readings in Culture, Gender, and Technology*. Indiana University Press, Bloomington, Indiana, 1991.

[11] Ross, J. *Image of Computing*. Available from http://www.imageofcomputing.com (2007); accessed August 31, 2008.

[12] Schneider, D. J. *The Psychology of Stereotyping*. Guilford Press, New York, 2004.

[13] Turkle, S. Computational Reticence: Why Women Fear the Intimate Machine. In *Sex/Machine*. Indiana University Press, Bloomington, Indiana, 1988.

[14] Turkle, S. and Papert, S. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society*, 16, 1 1990, 128-157.

Helping Students Debug Concurrent Programs

Jan Lönnberg
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland
jlonnber@cs.hut.fi

Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland
lma@cs.hut.fi

Anders Berglund[*]
Department of Information Technology
Uppsala Computing Education Research Group, UpCERG
Uppsala University
Uppsala, Sweden
anders.berglund@it.uu.se

ABSTRACT

We use empirical studies of how students understand concurrent programming and write concurrent programs to determine problem areas in students' understandings and approaches. We then suggest ways to deal with these problems to help students understand what is wrong with their concurrent programs. These include testing and visual debugging tools to help students find and understand their errors as well as feedback from teachers that makes use of these tools and knowledge of the students' understandings to clearly explain to students where they have gone wrong.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*; D.1.3 [**Programming techniques**]: Concurrent programming; D.2.5 [**Software engineering**]: Testing and debugging—*Debugging aids*

General Terms

Human Factors, Verification

Keywords

programming education, software visualisation

1. RESEARCH AIMS

The long-range goal of our project is to help programmers produce better concurrent programs. We approach this by developing methods and tools to help programmers understand what a concurrent program does. Better understanding is essential for both debugging and learning purposes.

[*]Currently also affiliated with Department of Computer Science and Engineering, Helsinki University of Technology

Our large-scale approach is to identify the needs of the intended users and then design solutions to address them. The general questions we therefore seek answers to are:

1. What kind of defects do programmers inexperienced in concurrent programming introduce in their concurrent programs, and why?
2. Which of these defects are hard to find or understand and why?
3. What kind of tools can assist a programmer in finding and understanding these most problematic defects, and how well do they work?

We focus on inexperienced users, in particular students, for several reasons. One is that their inexperience means they need more help. Another is that helping students understand their mistakes not only helps them get their programs to work; it also helps them learn. Finally, it is easier to introduce new ways to work to students than to experienced professionals with ingrained habits.

Our primary approach to helping programmers understand their programs is *software visualisation*; using a variety of presentation techniques to facilitate understanding of programs and algorithms and their behaviour. Price et al. [13] state that while software visualisation (SV) "has tremendous potential to aid in the understanding of concurrent programs", few SV systems have seen production use, especially in the domain of tools for professional programmers. They also note that when an SV system is designed, the content to be shown must be selected according to the goals of the system, which, in turn, are based on the requirements of the users. Thus, in order to answer our third question properly, we must answer the first two.

In this paper, we summarise some answers to the first two questions and use these as a basis for suggesting some for the third question. We also present questions about teaching programming raised when seeking answers to our questions.

2. STUDENTS' UNDERSTANDINGS

Defects are the result of a programmer's error. Many of these errors are due to wrong or incomplete understandings.

2.1 Understanding Program Execution

One thing that can be incorrectly understood is how a program is executed. Ben-David Kolikant finds that students initially approach a concurrent programming assignment from a user's perspective, in which program behaviour is seen only through the user interface, and not all of them are able to switch to a programmer's perspective [2]. Similarly, Ben-Ari and Ben-David Kolikant describe how high-school students make assumptions based on informal concepts rather than use formal rules and avoid using concurrency [1]. We found students who saw the programming assignment as an ideal problem, in which many limitations of real-life programming, such as finite memory or network delays, do not apply [11]. We also found that students introduce many defects in their programs that appear to be caused by misunderstanding or reasoning incorrectly about concurrent program execution [9].

It seems that part of the problem is that the program's runtime behaviour, a necessary part of the programmer's perspective, is hard to examine or interpret, preventing students from effectively understanding what their program does and reasoning in terms of the relevant concurrency model. This suggests to us that students need to be shown the consequences of their understandings of what their program is supposed to do, the circumstances it is supposed to work in and what correctness entails. Showing students the exact behaviour of a concurrent program is a complex issue that we discuss further in Subsection 4.1.

Providing students with tools to study memory allocation would help them understand how their programs use (or misuse) memory. In its most basic form, this would involve using a profiler to get information on the maximum memory use of their program. More detailed visualisations, such as charts that show memory use over time categorised by where the memory is allocated, can be used to help students understand memory use in more detail. Other resource usage issues, such as use of CPU time or network or disk capacity, can be handled similarly.

2.2 Understanding Goals

Students may also have a different understanding of what they are trying to achieve than their teachers. Ben-David Kolikant explains that students define a "correct program" as a program that exhibits "reasonable I/O for many legal inputs" [3]. We found that, apart from the expected problems with writing reliable concurrent programs, a lot of students wrote programs that were missing required functionality or implemented this functionality in ways that conflicted with requirements or required additional limitations on the runtime environment [9]. One reason we found for this was that students had different aims in their assignment, seeing it primarily as something they have to do to get a grade or as an ideal problem in an ideal context in which simplifying assumptions apply [11]. Others considered their submission a working solution to a real problem or even something that raises possibilities for future development [11]. The students also considered different potential sources of problems: the hypothetical user of the program (even when the assignment was specified in terms of the input and output of methods, not user requirements), underlying systems that could fail, especially network connections in a distributed system, and the programmer (the student) as a error-prone human [11].

The purposes of the programming task and sources of failure we found suggest that many of the errors made by students are misunderstandings of what their program is supposed to do and what situations it's expected to cope with rather than actual misunderstandings of concurrent programming itself. This is in line with our quantitative analysis of students' defects [9]. Assuming that it is desirable to have clearly-defined and specific goals (which is useful in guiding students' learning and simplifies assessment), this suggests that teachers should make goals more explicit and concrete. The goals should specify *what* the student should achieve rather than *how*, allowing students to find their own solutions to problems. Students should also be provided with ways to explore problems related to these goals. The student should see his or her program clearly fail to work correctly rather than be told afterwards that he or she did something the wrong way.

3. STUDENTS' DEVELOPMENT APPROACHES

Developing a concurrent program is a complicated business, and students are likely to go about it the wrong way.

3.1 Structuring the Solution

If we, as teachers or tool developers, are to communicate with students effectively, we need to speak their language. We found that students see the process of developing a program in a concurrent programming assignment in three different ways: writing the code that implements the solution, designing an algorithm that solves a computational problem and producing an application that solves a real-life problem. Similarly, they understood the tuple space data structure in four distinct ways, as a specification that describes the externally visible properties of the operations on the space, as an implementation of the tuple space described in terms of how it works, in terms of how it is used in a program and as one way out of many to achieve a goal in a program. In both cases, this shows that even in a very simple assignment, students can make use of different levels of abstraction to structure their solution. [10]

In Subsection 4.1, we discuss how this affects showing a student how his or her program works.

3.2 Finding Ways a Program Can Fail

Verification is the process of making sure a program is correct by finding any defects that may exist. The usual way to do this is to test the program. Finding sufficient test cases for a sequential program can be difficult. Exposing concurrency-related defects through testing is even harder.

Students are often quite sure of the correctness of their program and neglect to test it. In Ben-David Kolikant's study of high-school and college students who had finished their CS studies [3], more than a third of the students were sometimes satisfied with only compiling their program to ensure it is correct and half of them did not check that their program's output is correct.

We found our students have a wide range of approaches to testing. Some students used completely unplanned, cursory, testing. Some tried to 'break' the system (e.g. through stress testing), while others covered a variety of different cases. Moreover, some students found they cannot test their program adequately by themselves and need help from another person or tool, that testing in itself is not sufficient or that you have to prove your program correct yourself. [11]

The students' verification approaches could be improved by providing testing tools to generate scenarios that are hard to discover using normal testing procedures and more explicit and detailed guidance on how to apply different verification techniques in practice. The assignment itself could be changed to encourage students to learn and apply different verification techniques by explicitly requiring models, as done by Brabrand [4], or by requiring students to create suitable tests, e.g. using test-driven development.

Model checkers, such as Java PathFinder [16] are often used to find concurrency-related defects by specifying requirements that are checked against all the possible states of the program. If the requirement does not hold, a counterexample is generated that consists of an execution of the program that violates the requirement. Model checking can in many cases be used to prove correctness properties. However, as model checkers require that the program has quite a small state space and does not interact with entities outside the program model, adapting programs to a model that can be verified is often hard and error-prone work.

An alternative approach to finding concurrency bugs is to increase the chance of interleavings that lead to failure. Stress testing is a well-known approach, and its usefulness can be further improved by making sure interleavings occur often and in many places. One straightforward and realistic approach is to distribute the program's threads over multiple processors. Another way to do this is to automatically and randomly change the thread scheduling to make concurrency failures more likely to occur (e.g. [15]). This is the approach used by the automated testing system of our concurrent programming course to increase test effectiveness.

4. UNDERSTANDING FAILURES

Once a failure has been found, the underlying defect must be tracked down. Programmers can be helped to understand what their programs are doing by providing ways for them to explore their programs and by explaining their defects.

4.1 Software Visualisation

The goal of *Software visualisation* is to explain, through graphical representations, what a program does. Visualisation has been applied to debuggers to create *visual debuggers* such as DDD [18]; debuggers with graphical representations of data. Most debuggers concentrate on individual threads and can only show the current state of the program while the cause of a program malfunction usually lies in the past (which is especially problematic in concurrent programs, as duplicating a failure may be difficult); RetroVue [5], with its tree view of all executed operations, ability to examine all previous states of the program and thread display showing lock operations and execution times of threads, is a clear exception to this. However, it does not aid the programmer much in finding interrelated operations.

Answering queries about the reasons for events and states in a program is a promising new idea of this type that has not yet been developed to a level at which it can be used in concurrent software development. This approach has an obvious application in explaining to students how their programs failed. The Whyline [7], which uses dynamic dependence graphs to explain to novices the reason why a program did something (wrong), addresses the problem of explaining relationships. This approach has been found useful in some types of debugging situations in educational visual programming environments [7]. DDGs are of particular interest for concurrent programs, as interactions between threads (e.g. use of locking or shared variables) are clearly shown as edges.

A few debuggers and software visualisation systems have been designed with concurrency in mind. Most of them use sequence diagrams to display method calls; JaVis [12] adds collaboration diagrams to show interactions between objects. These diagrams can become cumbersome for complex executions. Kraemer [8] describes many visualisations for specific aspects of concurrent programs such as call graphs and time-process diagrams for message traffic. Visualisation techniques for programs can also be applied to study model checkers' counterexamples, as in e.g. Bogor [14].

Showing the user the executed instructions helps the user understand what the program is doing. In particular, showing the user the sequence of instructions that led to an unexpected event can be very useful; studies show that programmers often require information on the causes of an event and how different events are interconnected when looking for hard-to-find bugs [17, 6]. Concurrency can also make it very hard to trace the cause of an unexpected event.

Based on the empirical results above, we suggest that what is needed is a tool to generate execution history visualisations automatically from a running program that are easy to understand and navigate and provide the information needed by the student in an easily understandable form. Traditional visualisations such as sequence or collaboration diagrams are obvious approaches to doing this, and dynamic dependence graphs are a promising new addition.

There are several indications that students understand their program in terms of higher-level constructs than those in the code. One is that students see developing a program as solving algorithmic problems rather than straightforward implementation. Another is that they understand tuple spaces at higher levels of abstraction than their actual implementation [10]. This suggests that program visualisation tools should allow users to choose a level of abstraction by grouping together parts of the code or execution to correspond to their understandings, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. [13]. The tool could then visualise the behaviour of the program in a fashion closer to the student's view. For example, if students understand their programs as sets of communicating entities, the tool should be able to show them the communication between these entities and the relevant aspects of their state even though this state may be spread out over several objects, and part of the communication is implicit in locking mechanisms.

4.2 Feedback

Based on the reasoning above, we summarise our suggestions and propose a systematic way of providing feedback about programming assignments to students based on an understanding of the underlying misunderstandings (a similar format could be used in a more general programming context for bug reports):

1. An execution of the program which fails. This can be automatically generated by testing or model checking and shown using the visualisations we have described. The requirements of the assignment should be such that not adhering to them causes the program to fail in some situation the student can reconstruct. If the failure is incorrect behaviour (e.g. output), show it and the sequence of events leading up to it, focusing on the information relevant to the student understanding the failure. If the failure is resource overuse (e.g. memory), show how the resource is used (when and where).

2. A description of the defect that causes the failure. This can be expressed as a change to the code that eliminates the defect. Apart from failing executions, possible defects can in many cases be automatically listed based on empirical information on defects and the failures they result in. However, determining the exact defect will in most cases involve manual debugging work.

3. The underlying error. Using empirical information on the reasoning behind similar defects, and available information on the student's reasoning (e.g. documentation, comments, structure of code), a teacher can describe what he or she thinks the error is.

4. A teacher can try to determine what the student has not understood well enough and explain it.

5. Suggestions for how to detect similar problems: verification strategies effective against this type of defect and design strategies that avoid introducing them.

5. DISCUSSION

While these questions have been raised in a concurrent programming context, they are also interesting in a purely sequential context. However, the answers depend on whether concurrency is involved, particularly in the third question.

1. How explicit should teachers make assignment goals?

2. How should students be shown programs' resource use?

3. How should teachers encourage students to apply more useful ways of structuring and verifying programs?

4. How much help should students get in finding their programming errors?

6. REFERENCES

[1] M. Ben-Ari and Y. Ben-David Kolikant. Thinking parallel: The process of learning concurrency. In *Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 13–16, Cracow, Poland, 1999.

[2] Y. Ben-David Kolikant. Learning concurrency as an entry point to the community of computer science practitioners. *Journal of Computers in Mathematics and Science Teaching*, 23(1):21–46, 2004.

[3] Y. Ben-David Kolikant. Students' alternative standards for correctness. In *The Proceedings of the First International Computing Education Research Workshop*, pages 37–46, 2005.

[4] C. Brabrand. Constructive alignment for teaching model-based design for concurrency. In *Proc. 2nd Workshop on Teaching Concurrency (TeaConc '07)*, Siedlce, Poland, June 2007.

[5] J. Callaway. Visualization of threads in a running Java program. Master's thesis, University of California, June 2002.

[6] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997.

[7] A. J. Ko and B. A. Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the 2004 conference on Human factors in computing systems*, pages 151–158. ACM Press, 2004.

[8] E. Kraemer. Visualizing concurrent programs. In *Software Visualization: Programming as a Multimedia Experience*, chapter 17, pages 237–256. MIT Press, Cambridge, MA, 1998.

[9] J. Lönnberg. Student errors in concurrent programming assignments. In A. Berglund and M. Wiggberg, editors, *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling 2006*, pages 145–146, Uppsala, Sweden, 2007. Uppsala University.

[10] J. Lönnberg and A. Berglund. Students' understandings of concurrent programming. In R. Lister and Simon, editors, *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *Conferences in Research and Practice in Information Technology*, pages 77–86, Koli, Finland, 2008. Australian Computer Society.

[11] J. Lönnberg, A. Berglund, and L. Malmi. How students develop concurrent programs. In M. Hamilton and T. Clear, editors, *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, volume 95 of *Conferences in Research and Practice in Information Technology*, Wellington, New Zealand, 2009. Australian Computer Society. To appear.

[12] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In S. Diehl, editor, *Software Visualization*, pages 163–175, Dagstuhl Castle, Germany, 2002. Springer-Verlag.

[13] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[14] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: A flexible framework for creating software model checkers. In *Proceedings of Testing: Academic & Industrial Conference — Practice And Research Techniques*, June 2006.

[15] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

[16] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.

[17] A. von Mayrhauser and A. M. Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179, New York, NY, USA, 1997. ACM Press.

[18] A. Zeller. Animating data structures in DDD. In *The proceedings of the First Program Visualization Workshop – PVW 2000*, pages 69–78, Porvoo, Finland, 2001. University of Joensuu.

Minority Report

Computer Science Skills Perceived by Students in Different Disciplines

Tuukka Ahoniemi
Department of Software Sciences
Tampere University of Technology
Finland
tuukka.ahoniemi@tut.fi

ABSTRACT

Software skills are more and more required within different technical disciplines and the skill requirements vary through time and discipline. Within the Computer Science (CS) discipline the evolution is naturally rather well understood and adapted by Computer Science teachers. However the needs of other disciplines towards CS should be inquired from the representatives of those disciplines. Students themselves form an important part of it. This article presents excerpts from a research that identified differences between software skill requirements in other disciplines than Computer Sciences, or actually Information Teachnology in general.

The research was conducted by analyzing over 200 student responses to a questionnaire. The analyzed results show clear differences between the role of CS in different disciplines. This paper discusses the different roles of CS and poses new questions that should be considered when planning a CS curricula. The presented results also provide a basis for extending the research into a more general view.

Categories and Subject Descriptors

K.2.1 [**Computing Milieux**]: COMPUTERS AND EDUCATION—*Computer and Information Science Education*

General Terms

Human Factors

Keywords

CS Minor Student, Disciplines, Teaching, CS Curricula

1. INTRODUCTION

The rapid evolvement of computer sciences has naturally had an affect on the teaching of it, both pedagogically and contentually [5]. The pedagogical development is often something that is done between the teachers of that discipline. However, when it comes to the contents of teaching, the representatives of other disciplines should be listened also, especially if one is teaching CS for students majoring in something else than CS.

A wide needs assessment study on CS skills has been conducted by Sami Surakka [8]. In his thesis, Surakka evaluates the needs for different skills in the perspective of university curricula for students majoring in CS. In contrast to that, the research presented here leaves CS students (or actually all IT students) totally without attention.

This paper presents excerpts from a wider research that studied how non-IT students felt about the importance of different software skills [1]. The research also studied the opinions of teaching staff, but this paper is focused on students. These results form a basis for discussion on CS minor student motivation and on the contents of CS curricula.

To begin with, the paper introduces some motivation and background for the whole study. Then, the conducted research's settings are shortly introduced in Section 3 and its results in Section 4. Section 5 then discusses the results doing some conclusions about them, resulting into the posing of new, relevant questions.

2. BACKGROUND

As CS itself is something to study for CS students, it often is merely a tool for learning something else in other disciplines. For instance CS students learn SQL because they might need to be working with all sorts of databases in their future. An automation engineering student might need to learn SQL because some certain CAD-tool might the input of it and thus SQL is just a tool for learning something[1] to do with a CAD-tool. The approaches for learning SQL can then be really different between these two students.

Different approaches cause different motivations towards the subject. The possibility of using realistic examples that are related to one's own area of expertise has a notable impact on the motivation [2]. For instance, in the basic programming courses a problem of both parties getting frustrated has been observed by Wilson et al.[9]: Students studying something else than the programming itself do not necessarily yet understand they might need the general principles of programming in their later studies or work. On the other hand the ones studying programming do not understand the content of other disciplines well enough for the teacher to apply realistic programming examples. Different solutions have been applied to this problem by considering

[1]The author, having been only a CS student, does not know what could one do with such...

the creation of different programming courses for different disciplines.

For scientists a programming course that has both programming and science taught paraller with *breadth first* approach has been developed[4]. Using the same principles courses have been developed for art students [7]. Many experiments however are mostly based on things outside programming, for instance the use of games [3], and not the teaching of programming skills itself [4].

In high school level the results of teaching programming have been improved by choosing a language that is more easily approachable, like Python [6]. The effective use of teaching technologies like visualizations and pedagogical interpreters can also be used to facilitate the approach towards programming.

3. RESEARCH SETTINGS

The student opinions were measured using a quantitative research method, a questionnaire. When it comes to measuring opinion, a qualitative research method could have given richer results, but would not have been practical to apply. With a questionnaire it was possible to reach multiple students within each different discipline.

Not all students in all disciplines study CS, so it was important to select the students that do. To achieve this, the students were selected from those who had already enrolled to CS courses excluding the very first programming course. This ensured that all of the students had already studied at least the first programming course and knew something about CS, and of course were about to study more of it. Altogether 507 students were approached to answer the web questionnaire explained more thoroughly in the following.

3.1 The Questionnaires

The students were divided into two groups and both groups had their own questionnaire to answer: questionnaires A and B. The questionnaire A was wider in subjects, containing skills all around CS while questionnaire B was concentrated mostly on basic, fundamental skills of CS.

The division to two groups was based on the amount of CS studies the student had completed. The purpose was not to have second-year students answering a questionnaire full of skills they have not yet heard of. The students were allowed to make the decision by themselves. They had both of the questionnaires available and they were instructed to choose questionnaire B if they had been studying only "couple of" courses of CS and questionnaire B if they had studied more.

The questionnaires contained a categorized list of skills, each related to the question: *"How important do you find the following skills concerning your future (work and studies)?* The students valued each skill with an integer out of scale 1 to 4. 1 meaning completely unimportant and 4 meaning very important. If a skill or its importance was unknown for the student he could choose option "I do not know".

All the different skills that were asked to rate are not explained here but in general the students rated all sorts of skills related to the following categories: Basic programming (only in questionnaire B), Object-Oriented Programming, Data Structures and Algorithms, Software Process and Different Environments and Platforms. The results section presents some of the most interesting skills individually also. The results were analyzed calculating weighted averages for each disciplines and then comparing the answers.

3.2 Degree programmes under focus

Table 1: Degree programmes under focus and their abbreviations used in this paper

Abbrv.	Degree programme
AU	Automation Engineering
EL	Electrical Engineering
SC	Science and Engineering
IKM	Information and Knowledge Management
COM	Communications and Electronics

The original research was focused to cover all different disciplines inside Tampere University of Technology (TUT) except Information Technology. However, this paper presents only the most important results from the the most interesting disciplines. Because different universities have degree programmes of their own it is needed to describe briefly the degree programmes that are discussed here.

Table 1 presents the degree programmes and their abbreviations that are used later in this paper when presenting the results. Besides more traditional degree programmes like Automation Engineering (AE), Electrical Engineering (EL) and Science and Engineering (SCI) this study includes Information and Knowledge Management (IKM) and Communications and Electronics (COM).

Information and Knowledge Management is a proportionally new degree programme in TUT which concentrates on handling and managing rapidly growing amount of information training specialists that understand both information technology and industrial engineering. The studies include a good overall view on software engineering also, but the focus is not at all on developing software. This makes IKM students an interesting group from the perspective of CS teacher. The students have totally different background and basis than CS students.

Communications and Electronics, roughly said, is a combination of information technology and electronics. COM students study plenty of software skills and like IKM students, their focus is different than CS students. COM students use software skills as a tool for creating networks and network communication as well as creating hardware and systems in general–software skills are often used in a lot lower abstraction level and in more restricted programming environments.

4. RESULTS

This section presents main output of the student questionnaires. The values of separate skills are left out, but the most interesting differences are mentioned to form the basis for the discussion. Some of the following skill categories also include numerical representation of the weighted averages of all the skills within that category–to clarify the differences amongst the degree programmes in a wider perspective. The scale is from 1 to 4, 1 meaning a completely useless skill and 4 the most important value.

Altogether 256 responds were received out of 507, so 50% of the students answered the questionnaire. For the disciplines that are under the focus of this research, a total amount of 231 responds were received.

4.1 Basic programming skills

All the skills in this category were included only in the B questionnaire. This means that only students having studied only a course or two of computer sciences rated these.

Basic programming skills were perceived important in general. The average for all the skills by all the respondents was 3.0. The one skill being noticeable less important than other skills was the most theoretical one, *recursion.* The average values for all the disciplines can be seen from Table 2.

A big exception within this category was the overall opinion of students inside Information and Knowledge Management degree programme. As all other disciplines valued the skills of this category with an overall average at least 3.0, IKM students gave only an average value of 2.2.

Table 2: The average importance of all basic programming skills in different disciplines

Programme	AU	EL	SC	IKM	COM	All
N (only quest B)	32	26	27	21	27	133
AVG	3.1	3.2	3.0	2.2	3.3	3.0

4.2 Object-Oriented Programming and Design

In general the skills within this category were found important: The average of all the skills within all the respondents was 3.0. Especially the skill *basics of object-oriented programming* were found very important by all disciplines.

The respondents of A-category valued all of the skills in this category at least quite important without big differences between disciplines. Science and Engineering students, however, found *design patterns* and *software architectures* less important than others.

The IKM students who answered B-questionnaire gave noticeable smaller values than respondents from other disciplines. For example, for *inheritance*, IKM students even gave the only non-important value in this whole category: only 1.8 whilst the average of all respondents is 2.6.

4.3 Data Structures and Algorithms

Table 3 presents the perceived importance of data structures and algorithms in different disciplines. As expected, there were differences between the importance of different skills within this category. The average of all skills by all respondents is the scale middle 2.5. The most important skills were *the selection of a suitable data structure* and *the selection of a suitable algorithm.* These were found important by nearly all disciplines.

Again, the Information and Knowledge Management students who answered questionnaire B were the most critical towards the importance of the skills within this category. This supports the view of the IKM degree programme teaching staff mentioned in the wider research[1]: The IKM students dread the data structures and algorithms courses because they feel the courses are too technical for their focus.

Table 3: The average importance of all data structures and algorithms related skills in different disciplines

Programme	AU	EL	SC	IKM	COM	All
N	62	50	32	37	50	231
AVG	2.5	2.5	2.9	2.2	2.7	2.5

4.4 Software Process

Table 4 presents the perceived average importances of software process related skills. The average of all skills by all respondents was as high as 3.3 meaning the Software Process in general is important for all the students. Extremely important skills were *understanding the program specification document* and *understanding the whole software process.* Especially Information and Knowledge Management students found these two very important giving as high average values as 3.9.

The students in Science and Engineering degree programme gave noticeable smaller values to all of the skills. Students in other disciplines on the other hand valued the skills high. Even the advanced skills like *creating a program design document* and *software project management* were valued important or even very important.

Table 4: The average importance of all software process related skills in different disciplines

Programme	AU	EL	SC	IKM	COM	All
N	62	50	32	37	50	231
AVG	3.4	3.3	2.3	3.5	3.4	3.3

4.5 Different Environments and Platforms

This category consisted of many various skills and thus it is irrelevant to handle this category as a whole instead of single skills. The most important of them were *robust programs* (average 3.2) and *basic problems and concepts of concurrency* (average 3.1 in questionnaire A, only 2.2 in questionnaire B, though). The least important were *computer graphics* and *programming for mobile platforms.*

Between different disciplines there were differences in the results of this category. The Information and Knowledge Management students who answered questionnaire B gave Basic Problems and Concepts of Concurrency only a really low value of 1.5 while the average of all respondents was 2.2 and Science and Engineering students gave as high value as 3.3. This result however does not give anything generalizable about the disciplines because the answers to questionnaire A in these two disciplines are completely the other way around. What this can tell about is the differences between one discipline: Students who study more Software Sciences can have a completely different target for the software skills. One must also notice the small amount of SCI respondents to questionnaire A: Only five answered were received.

Automation Engineering, Electrics, and Communications and Electrics students perceived these skills rather important in general. Especially lower abstraction level programming skills like Low-Level Programming and Embedded Systems were found important within these disciplines.

5. DISCUSSION AND CONCLUSIONS

It is of course relevant to question that *"Do students really know what they need? Do they know what is best for them? Aren't teachers just for that reason–to know better?"* Yes, teachers know better, and the problem with these results would also be that some students know better than others and thus the ones that do not know would skew the results. However, this is not the problem in this context. The results are still relevant.

How important students perceive a subject for their own future, is directly related to their motivation. To avoid major lacks of knowledge due to this lack of motivation, a set of compulsory courses has been ruled within each degree programme. Still, the motivation problem affects the studying performances in spite of the obligatoriness of the course. For voluntery courses the perceived importance affects the enrollment directly. So at the end, the results of this paper can be treated as a some sort of a motivation meter.

The research gave expected results: There are differences between disciplines when CS skills are concerned. This is rather obvious but the idea was also to find out what sort of differences they are.

On the basis of these results (and the wider ones including the teaching staff view [1]) within the disciplines Automation Engineering, Electrics, and Communications and Electrics software skills act a great role–work related to own discipline subject requires the applying of software skills as a tool. This applying is often related to the more technical side of computer sciences: programming, and especially programming to more restricted, embedded systems.

Information and Knowledge Management and Science and Engineering students represent a completely different approach to CS needs. IKM students focus more on the abstract, software process related skills. SCI students on the other hand do not find software process relevant at all but use CS mostly as a tool for mathematical modeling.

A positive result was that nearly all students found basic programming skills important. An interesting exception for this was the IKM students answering to questionnaire B. They found only few software skills important despite the big role of CS in the general idea of the whole degree programme. An easy explanation to this is again to agree with the previous questioning: The students might not yet know what they really need. This opinion is supported by the fact that IKM students having studied CS a bit further (questionnaire A respondents) found the skills much more important. Another explanation is the structure of CS studies for IKM students within the curricula of Tampere University of Technology[1].

The respondents all represented students of same university which greatly weakens the generalizability of the results. Once again, the results thus have to be treated as interesting and suggesting. Another important factor weakening this aspect is that no advanced statistical analysis was made for the results.

A proper statistical analysis of the results would have required a greater amount of respondents and even with that would have still had the following lacks considering generalizability:

- Respondents come from one university and represent only one nationality (characteristics of disciplines and future work life expectations could vary locally).
- The answers would still represent only the opinions of students. The opinions of teaching staff and persons from business life should also be taken into account.

Expanding and internationalizing the amount of respondents is something that could be done in the future and the opinion of teaching staff is touched upon in the larger research [1]. Finding enough respondents from business life that would represent the needs of a certain discipline students is then almost an impossible task. But having completely generalizable results is not actually that important because nevertheless the universities that could apply the results have that much of variance within, so that the applying would anyway require adaptation. For further studies, this research should offer ground work to continue with.

These results represent the view of students and by adapting that, it is possible to benefit out of the results. The results raise the following questions that should be considered inside other universities/institutes of teaching also, especially when designing a CS curricula:

- Is the CS curricula done only considering CS students or does it also take into account students doing only a CS minor?
- Is the set of courses that is offered as *The CS minor degree* suitable for all the students in different disciplines that actually need CS? Or should multiple different CS minor degrees be offered? For instance Communication and Electrics students would benefit from low abstraction level courses with more technical aspect where as Information and Knowledge Management students need more overview-like courses and high abstraction level courses more related to running a software project.
- And most of all: How could we fix the possible gap between student and teaching staff opinions on what really is important? *How could the students know better what is best for them?*

6. REFERENCES

[1] T. Ahoniemi. Ohjelmistotekniikka eri koulutusohjelmissa. Master's thesis, Tampere University of Technology, 2008.

[2] T. Ahoniemi, E. Lahtinen, and K. Valaskala. Why Should We Bore Students When Teaching CS? In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research*, November 2007.

[3] J. D. Bayliss and S. Strout. Games as a "Flavor" of CS1. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 500–504, New York, NY, USA, 2006. ACM.

[4] Z. Dodds, C. Alvarado, G. Kuenning, and R. Libeskind-Hadas. Breadth-First CS 1 for Scientists. *SIGCSE Bull.*, 39(3):23–27, 2007.

[5] G. Engel and E. R. (Eds.). ACM Computing Curricula 2001. Computer Science. 2001.

[6] L. Grandell, M. Peltomäki, R.-J. Back, and T. Salakoski. Why Complicate Things?: Introducing Programming in High School using Python. In *ACE '06: Proceedings of the 8th Australasian conference on Computing education*, pages 71–80, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[7] L. A. S. King and J. Barr. Computer Science for the Artist. *SIGCSE Bull.*, 29(1):150–153, 1997.

[8] S. Surakka. *Needs Assessment of Software Systems*. PhD thesis, Teknillinen korkeakoulu, 2005.

[9] G. Wilson, C. Alvarado, J. Campbell, R. Landau, and R. Sedgewick. CS-1 for Scientists. *SIGCSE Bull.*, 40(1):36–37, 2008.

Student-Generated Podcasts for Learning and Assessment

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

ABSTRACT

The aim of this paper is to discuss our experience with, and some broader thoughts on, the use of student-produced podcasts as a means of supporting and assessing learning. The results of an assessment using this medium are reported, and student evaluation of the assessment presented and discussed.

1. INTRODUCTION

The aim of this paper is to discuss our experiences with using student-produced podcasts as a means of assessment in computer science.

Podcasts, and related media such as radio programmes, are an increasingly important way of communicating science to a general audience. Using podcasts as an assessment method presents an opportunity for students to engage with course material in a fresh new medium.

A number of projects have tackled aspects of university learning using podcasts. Most commonly, these are used as ways of presenting course material. For example, the *ProfCast* software (`www.profcast.com`) is widely used in many universities to record and make available lecture material. A large amount of advocacy has been made concerning this teaching style. However, only a small number of studies (e.g. [12, 8, 9]) have attempted to evaluate the effects on learning. Some of the more substantial studies include those of Evans[7], Bell et al. [2], Edirsingha&Salmon [6], and Baird&Fisher [1]. These generally portray a positive picture of such use. However, these studies tend to show this solely from the point of view of students' self-evaluation, which is valuable but provides only a single perspective.

Another way in which podcasts have been used is in providing short podcasts that give supplementary material to students; such attempts have been analysed by Clark, Taylor et al. [4, 5], who give a largely positive summary of the benefits. A second use of short podcasts is in providing assessment feedback [11]. Bennedsen and Caspersen [3] have used video podcasts of the program development process to act as a demonstration of practical skill.

However, most efforts in using podcasting in education have been teacher-created. In this paper we explore the idea of *student*-created podcasts for learning and assessment. This has received little attention in the research literature. One recent example is given by Thompson [13], who discusses the use of student-created podcasts as a way for teacher-training students to reflect on their learning and to provide a repertory of teaching tips to be shared amongst students on a course.

2. THE ASSESSMENT

We prepared a podcast-based assessment as a component (5% of total module marks) of the assessment on our *Introduction to Intelligent Systems* module. This is typically taken by students mid-way through their degree, and is typically taken by 40–50 students. There were a number of reasons for adopting this form of assessment:

- We hypothesised that students would find an alternative form of assessment interesting and fresh, and would engage student interest more than an assessment in a previously encountered style. In particular, we believed that students at this stage in their degree might be revitalised by being presented with a novel form of assessment.
- We wanted students to begin to engage with the research literature. However, at their stage of subject knowledge, asking detailed questions requiring technical knowledge of research literature is too advanced. Therefore, this was seen as a way of getting students to engage with the gist of some research papers, without having to go into technical detail. Therefore, this assessment acts as part of the students' *learning* on the course, as well as forming a summative *assessment*.
- We believe that it is an important skill for science students to be able to communicate about their work to a general audience, and this type of activity provides some early practice in this.

The assessment is given in figure 1. Students could work on this as individuals or in a group of two (in which case both students get the same mark). Students were also given advice about how to cite sources, and how to hand in the assessment. Beyond the advice given in the assessment, no specific guidance was given about the practical issues of creating the podcast; it was assumed that students doing a computing degree would have sufficient general IT knowledge to be able to work this out for themselves (students with practical issues were encouraged to contact the course tutor; no such queries were raised). If this kind of assessment were to be applied elsewhere, it would be probably be necessary to provide more detailed instructions/training in the use of recording software and equipment.

Question 2

Create a short (audio) "podcast" which gives an overview of some piece of research in neural networks. You should record a talk/discussion of around five minutes, which presents the main points of a neural network research paper of your choice in a style that would be accessible to a general audience with some broad scientific knowledge.

To find a paper to talk about, use a site such as `http://scholar.google.com`. You will probably find a topic about some application of neural networks to be most accessible. For example, you might use "neural networks" "face recognition" to find a paper about the application of neural network techniques in face recognition.

You can choose a topic that has been covered in the lectures, or another topic. Fewer marks will be available if you choose a topic that has already been covered in some detail in the lectures.

Please write the reference to your paper on the paper hand-in, using the format given in the section *how to quote bibliographical sources* below. If you use any additional resources (papers, textbooks, web sites) please also mention these.

Your podcast should be a single audio file of around 5 minutes. There will be a penalty for any files that run significantly over 7 minutes or under 3 minutes. Your file should be in .mp3, .wav, or .ogg format. Hardware for audio recording can be found in the multimedia room in the Octagon.

To give you an idea of the sort of thing that we are looking for, have a listen to the podcasts at:

- `http://www.bbc.co.uk/radio4/science/thematerialworld.shtml`
- `http://www.guardian.co.uk/science/podcast`
- `http://www.nature.com/nature/podcast/index.html`

Figure 1: Details of the podcast assessment.

3. ISSUES FROM THE ASSESSMENT

A number of issues arose from the assessment and from thinking around this kind of assessment in general.

3.1 Student Reaction

The first reaction we received to setting this assessment was a student asserting that the form of the assessment was "offensive" and "degrading". A couple of students also sought reassurances that the audio files would not be made available on the department website or to other students. The nub of this seems to be that the use of voice, as opposed to written material, has a "personal" quality to it, that is not an issue when it comes to other forms of presentation. In particular, the recorded voice has particular issues, as we are not accustomed to the sound of our recorded voice, and many people react negatively to hearing their recorded voice.

Other students informally expressed a positive attitude towards the assessment, in particular commenting on it being something interestingly different to what they usually do.

3.2 Unexpected Issues

A small number of unexpected issues arose as a result of the assessment:

Two students chose to submit work using a computer-synthesized voice, one explaining that they had attempted to record their voice and had not liked it, another submitting in this form without explanation.

One student "group" consisted of two students, but only one spoke on the recording.

Several students complained about the difficulty in finding relevant research papers, in particular ones that were available without charge, despite the advice given about finding papers. This was surprising, but might reflect (1) students working off-campus and not having automatic IP-related logins to certain university library subscription journals, or (2) weak web-search skills on behalf of the students.

3.3 Diversity Issues—Disability and Personality Diversity

This form of assessment could provide particular difficulties for students with certain disabilities, which do not occur as problems elsewhere in the range of assessments. We offered an alternative assessment to any students who were affected by this.

Another diversity related issue relates to the well-known idea that a wide range of assessment methodologies is a positive thing because it gives students with different preferences in styles of learning/presentation an opportunity to shine. Does this sort of assessment, for example, give an opportunity for students who are more fluent in speaking to be assessed using those skills, as opposed to the fluency in writing that is assessed in many assessments? Or is this diversity in preferences overemphasised?

3.4 Marking

One of the advantages of this as an assessment medium is that marking is very practical. It is possible to listen to the assessment whilst simultaneously writing comments. This presents a valuable practical advantage to this form of assessment, as we are are under increasing pressure to find forms of assessment that can be marked efficiently without compromising on the quality of evaluation or feedback given.

One issue of concern encountered during marking was that of form versus content. We decided not to specifically allocate marks to these two aspects of the assessment, as it is in practice difficult to separate them out. Whilst informal efforts were made to avoid being swayed by the presentational confidence of the students, there is a danger in marking this kind of work that a presentation presented confidently and fluently can have a spurious authority that a better-researched but shoddily presented piece of coursework does not have.

A few students submitted files that, despite claiming to be in one of the formats specified, did not play using standard software. Sorting out these issues took a lot of time.

One useful exercise for analysing an assessment is to note what comments were made repeatedly when marking the work. This can be usefully communicated back to the students for general feedback, and to future years of students as "common pitfalls". When marking this work, the following issues came up in a number of different stu-

Question	**1**	**2**	**3**	**4**	**5**	**Mean**	**StdDev**
Did you think that this was a useful assessment in terms of learning new material and presenting what you had learned? (1 (not useful)–5 (very useful))	2	1	3	4	3	3.4	1.3
Do you think that this is a kind of assessment that we should use in the future? (1 (not at all)–5 (yes, very much))	3	2	2	4	2	3.0	1.4
Was the assessment well explained? (1 (not at all well explained)–5 (very well explained))	0	2	5	3	3	3.5	1.0

Table 1: Numerical evaluation of the assessment: question, numbers of responses at each scale-point 1-5, mean, standard deviation.

dents' work:

- There is not enough structure to the talk; alternatively, there is structure, but the "scaffolding" language used to flag up the structure was not present.
- There were inconsistencies in the *granularity* of explanation throughout the talk. In particular, students would leap from highly detailed explanations of one component of the material, to very general explanations of a related part. Also, some weak assessments didn't show any sense of *direction* to the granularity: they might have been improved e.g. by starting with higher level explanations and then "drilling down" to more technical detail.
- There were problems with the use of technical vocabulary. Some students used a vocabulary that was far too advanced for the audience specified. Instead, they could either have defined technical terms in simpler language, or sometimes just avoided it and explained things directly in a simpler way.

By contrast, the following were positive features that appeared commonly in marking

- Well structured, and structure well explained.
- Clear explanation.
- Appropriate for the specified audience.

3.5 Evaluation

Students were asked to evaluate the podcast assessment in two ways: through three questions on a 5-point Likert scale, and through free text comments. The results from the numerical evaluation are given in table 1. Thirteen students responded. Overall these results show a very mixed view of the assessment.

The free text comments also showed a diversity of opinion about the value of the assessment. A number of students remarked positively on the originality of the method of assessment, and the ability to choose a topic freely within the scope of the module. However, a number of students expressed problems with knowing what assumptions should be made about the audience, about access to papers (as noted above) and finding relevant papers, and about the practicalities of recording and producing an effective podcast. A number of students suggested that a written alternative should have been offered, and commented on the lack of a detailed mark scheme.

4. CHANGES

There are a number of things that we might do differently if presenting a similar assessment in future years. In particular, we would consider:

- Give some more instructions about how to structure a presentation in this form. A number of podcasts submitted showed evidence that students had read and understood the material, but the actual presentation was weak. In particular, ways of marking out sections and providing a "scaffold" for the overall structure of the talk.
- Give more detailed instructions about how to find a relevant paper, in particular instructing the students that they might find more free-to-access papers by using a computer on the university campus rather than their computer at home.
- Providing more explicit guidance about the audience that the podcast should be targeted at; one way to do this would be to give particular exemplars of the kind of audience being targeted rather than a generic description.
- We are uncertain as to whether it would be sensible to divide the marks for form and content. Whilst this would potentially be valuable, it may prove difficult to do in practice.

5. QUESTIONS FOR DISCUSSION

- One argument for setting this kind of assessment is that a large proportion of students doing the assessment are "digital natives" [10], and are likely to relate to material such as podcasts rather than traditional forms of assessment such as essays. Is this really true? There does not appear to have been any academic work on the demographics of podcast listeners, and evidence from less formal surveys reported in the press appears to be inconclusive (see e.g. `http://www.comscore.com/press/release.asp?press=1438`,`http://www.eweek.com/c/a/Messaging-and-Collaboration/What-Blogs-Podcasts-Feeds-Mean-to-Bottom-Line/`, `http://www.vnunet.com/vnunet/news/2141338/ youth-today-spurn-podcasting`). How much do students expect university work to reflect the values of the "world outside" versus being an internal world with its own ways of doing things?
- Is it appropriate to expect students to "perform" in this fashion? Is it beyond the reasonable expectations of students that they are assessed using the medium of recorded voice? Is this too personal a medium to be used in assessment?
- Is there a demographic bias in the kind of students who listen to podcasts, and therefore a bias in the assumption that this is a more "native" form of assessment for most students. For example, some surveys

of podcast usage have suggested gender and age biases in general podcast listening (e.g. `http://www.comscore.com/press/release.asp?press=1438`). Is this an issue for the use of podcasts in learning?

- How can we separate form and content in marking this kind of assessment? Indeed, should we?
- Could we use these in a shared fashion, e.g. for sharing information between students? Would there be a way of introducing this so that students would find it acceptable?
- Is there a danger of the advantages of this being temporary? Is there a danger with these "gee-whiz" technologies just being seen as a vacuous attempt to "be trendy"?
- Is this a particularly good, or particularly bad, form of assessment for computer science students by contrast with students from other subjects?
- Would it be interesting to explore a multi-episode podcast, e.g. as a way of getting students to reflect on an ongoing piece of project work?; or, as a way of supporting student learning by asking them to produce a regular podcast covering various chapters of a book, a collection of research papers, or similar.

6. CONCLUSIONS

We have discussed our attempt at using student-generated podcasts as a way of promoting learning and of carrying out assessment. Overall, the reaction to this amongst students was mixed. We have presented a number of issues that arose during the development and marking of this assessment, and presented a number of questions for discussion and for refection by teachers who are planning to use this form of assessment themselves.

7. REFERENCES

[1] Derek E. Baird and Mercedes Fisher. Neomillennial user experience design strategies: Utilizing social networking media to support "always on" learning style. *Journal of Educational Techology Systems*, 34:1, 2005-06.

[2] T. Bell, A. Cockburn, A. Wingkvist, and R. Green. Podcasts as a supplement in tertiary education: an experiment wih two computer science courses. In *Proceedings of MoLTA 2007*, 2007.

[3] Jens Bennedsen and Michael E. Caspersen. Revealing the programming process. *SIGCSE Bull.*, 37(1):186–190, 2005.

[4] Steve Clark, Catherine Sutton-Brady, Karen M. Scott, and Lucy Taylor. Short podcasts: The impact on learning and teaching. In *Proceedings of mLearn 2007*, pages 285–289, 2007.

[5] Steve Clark, Lucy Taylor, and Mark Westcott. Using short podcasts to reinforce lectures. In *UniServe Science Teaching and Learning Research Proceedings*, pages 22–27, 2007.

[6] Palitha Edirsingha and Gilly K. Salmon. Pedagogical models for podcasts in higher education. In *Proceedings of the EDEN Conference*, 2007.

[7] Chris Evans. The effectiveness of m-learning in the form of podcast revision lectures in higher education. *Computers & Education*, 50(2):491–498, February 2008.

[8] Maree Gosper, Margot McNeill, Karen Woo, Rob Phillips, Greg Preston, and David Green. Web-based lecture recording technologies: Do students learn from them? In *Proceedings of EDUCAUSE Australasia 2007*, 2007.

[9] C. McLoughlin and M. Lee. Listen and learn: A systematic review of the evidence that podcasting supports learning in higher education. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, pages 1669–1677, 2007.

[10] Mark Prensky. Digital natives, digital immigrants. *On the Horizon*, 9:5, 2001.

[11] Chris Ribchester, Derek France, and Anne Wheeler. Podcasting: A tool for enhancing assessment feeedback. In *4th Conference on Education in a Changing Environment*. Salford University, September 2007.

[12] S.K.A. Soong, L.K. Chan, C. Cheers, and C. Hu. Impact of video recorded lectures among students. In *Australasian Society for Computers in Learning in Tertiary Education (ASCILITE) Conference 2006*, 2006.

[13] Linda Thompson. Podcasting: The ultimate learning experience and authentic assessment. In *ICT: Providing Choices for Learners and Learning. Proceedings of Asciilite Singapore 2007*, 2007.

Algorithm Recognition by Static Analysis and Its Application in Students' Submissions Assessment

Ahmad Taherkhani
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
ahmad@cs.hut.fi

Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
lma@cs.hut.fi

Ari Korhonen
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
archie@cs.hut.fi

ABSTRACT

Automatic program comprehension (PC) has been extensively studied for decades. It has been studied mainly from two different points of view: understanding the functionality of a program and understanding program structure. In this paper, we address the problem of automatic algorithm recognition and introduce a method based on static analysis to recognize algorithms. We discuss the applications of the method in the context of automatic assessment to widen the scope of programming assignments that can be checked automatically.

Keywords

program analysis, algorithm recognition, automatic grading

1. INTRODUCTION

Current automatic assessment systems for programming exercises, such as CourseMarker [9], BOSS [12] or Web-Cat [4] provide many advantages for large programming courses. These systems have versatile methods to check the submitted programs, among which checking correctness and functionality (tested against teacher's test data) and program structure are perhaps the most important ones.Other features available in some systems include checking programming style [5], the use of various language constructs [19], memory management [1], and run time efficiency [19].

Despite the multifunctional capabilities, it may be difficult to set up assignments that require the use of *specific* algorithms. For example, a data structures course could have assignments such as "Write a program that sorts the given array using merge sort", or "Write a program that stores the given keys in a binary search tree and outputs the keys in preorder". In this case, applying automatic assessment would require setting up tests for intermediate states of the algorithm instead of only the final state or output, yet the tests may not reveal usage of a different algorithm than requested.

In this paper, we present a method, based on research on PC, that supports automatic checking of and feedback on algorithms. The prototype implementation is based on applying static analysis of program code, including various statistics of language constructs and especially *roles of variables*. These allow the characteristics of various algorithms to be distinguished. The prototype has been built to distinguish between several common sorting algorithms, but the technique can be applied to other classes of algorithms as well. The first steps in our research on PC (in general) and a description of the new method is presented. Finally, some preliminary results and discussion follow.

2. PROGRAM COMPREHENSION

The problem of PC can roughly be classified into three categories. In the following, we present these categories and describe the most common applications of each.

Understanding functionality: Most of the studies in the field fall into this category. The PC problem is seen as the problem of understanding what the program does. The earlier studies were mainly motivated by the need of software developers, testers and maintainers to understand a system without having to read the code, which is a time-consuming and error-prone task (see for example [8, 14]). An automatic PC tool could be useful in software projects, for example, in verification and validation tasks.

Analysing structure and style: PC can be seen as examination of the source code, for example, to see how control structures are used and to investigate coding style. The objectives of these analyses could be to monitor students' progress, to ensure that students' submissions are in accordance with teachers' instructions, and to get a rough idea about the efficiency of the code. Tools that perform these kinds of analyses are mostly used in computer science-related courses at universities and are often integrated into plagiarism detection systems [5, 17, 18].

Recognizing and classifying algorithms: The PC problem can also be viewed as the problem of algorithm recognition, i.e., being able to classify algorithms implies understanding a program. Therefore, the process of finding out what family of algorithms an algorithm belongs to or what kind of algorithm it resembles involves program comprehension tasks. The primary use of such an algorithm recognition tool is to examine submitted exercises and make sure that students have used the algorithm that they have been asked to. In this paper, the PC is discussed from this point of view.

Methods in the PC field can be divided into two categories: dynamic and static analysis. In dynamic analysis, a program is executed by some input, and the output is investigated in order to understand the functionality of the program. These methods are often used in automatic assessment systems, where the correctness of students' submissions is tested by running their program using some predefined test input and comparing its output with the expected one (see for example [4, 9, 12]).

Static analysis, on the other hand, involves no execution of the code. These approaches analyze a program using structural analysis methods, which can be carried out in many different ways, focusing on different features of the code, for example, the control and data flow, the complexity of the program in terms of different metrics, etc. Most PC studies are based on static program analysis. We present the main approaches below.

2.1 Knowledge-based approaches

Knowledge-based techniques concentrate on discovering the functionality of a program. These approaches are based on a knowledge base that stores predefined plans. To understand a program, program code is matched against the plans. If there is a match, then we can say what the program does, since we know what the matched plans do. The plans can have other plans as their parts in a hierarchic manner. Depending on whether the recognition of the program starts with matching the higher-level or lower-level plans first, knowledge-based approaches can be further divided into three subcategories: *bottom-up*, *top-down*, and *hybrid* approaches.

Most knowledge-based approaches work bottom-up, in which we try to recognize and understand small pieces of code, i.e., basic plans first. After recognizing the basic plans, we can continue the process of recognizing and understanding higher-level plans by connecting the meanings of these already recognized basic plans and by reasoning what problem the combination of basic plans tries to solve. By continuing this process, we can finally try to conclude what the source code does as a whole. In top-down approaches, the idea is that by knowing the domain of our problem, we can select the right plans from the library that solve that particular problem and then compare the source code with these plans. If there is a match between source code and library plans, we can answer the question of what the program does. Since we have to know the domain, this approach requires the specification of the problem (see, for example, [10]). Hybrid approaches (see, e.g., [15]) use both techniques.

Knowledge-based approaches have been criticized for being able to process only toy programs. For each piece of code to be understood, there must be a plan in the plan library that recognizes it. This implies that the more comprehensive a PC tool is desired to be, the more plans must be added into the library. On the other hand, the more plans there are in the library, the more costly and inefficient the process of searching and matching will get. To address these issues of scalability and inefficiency, various improvements to these approaches have been suggested including fuzzy reasoning [3]. Instead of performing the exhaustive and costly task of comparing the code to all plans, fuzzy reasoning is used to select a set of more promising pieces of code, i.e., chunks, and carry out the matching process in more detail between those chunks and the corresponding plans.

2.2 Other approaches

The following approaches to PC can also be discerned.

Program similarity evaluation approaches: As the name suggests, program similarity evaluation techniques, i.e., plagiarism detection techniques are used to determine to what extent two given programs are the same. Therefore, these approaches focus on the structural analysis and the style of a program, rather than discovering its functionality. Based on how programs are analyzed, these approaches can be further divided into two subcategories: *attribute-counting* approaches [5, 17] and *structure-based* approaches [18]. In attribute-counting approaches, some distinguishing characteristics of the subject program code are counted and analyzed to find the similarity between the two programs, whereas in structure-based approaches the answer is sought by examining the structure of the code.

Reverse engineering approaches: Reverse engineering techniques are used to understand a system in order to recover its high-level design plans, create high-level documentation for it, rebuild it, extend its functionality, fix its faults, enhance its functions and so forth. By extracting the desired information out of complex systems, reverse engineering techniques provide software maintainers a way to understand complex systems, thus making maintenance tasks easier. Reverse engineering approaches have been criticized for the fact that they are not able to perform the task of PC and deriving abstract specifications from source code automatically, but they rather generate documentation that can help humans complete these tasks [16]. Since providing abstract specifications and creating documentation from source code are the main outcomes of reverse engineering techniques, these techniques can be regarded as analysis methods of system structure rather than understanding its functionality.

In addition to the aforementioned techniques, the following approaches to understand programs or discover similarities between them have also been presented: techniques used in clone detection methods [2], PC based on constraint satisfaction [21], task oriented PC [6] and data-centered PC [11]. Detailed discussion of these approaches is beyond the scope of this paper.

3. METHOD

Our approach in recognizing algorithms is based on examining the characteristics of them. By computing the distinguishing characteristics of an algorithm, we can compare these characteristics with those collected from already recognized algorithms and conclude if the algorithm falls into the same category.

We divided the characteristics of a program into the following two types: *numerical characteristics* and *descriptive characteristics*. Numerical characteristics are those that can be expressed as positive integers, whereas descriptive characteristics cannot. The numerical characteristics examined in our method are: number of blocks (NoB), number of loops (NoL), number of variables (NoV), number of assignment statements (NAS), lines of code (LoC), McCabe complexity (MCC) [13], total operators (N_1), total operands (N_2), unique operators (n_1), unique operands (n_2), program length (N = N_1 + N_2) and program vocabulary (n = n_1 + n_2). The abbreviation after each characteristic is used to refer to it in Table 1, which is explained later in this section. From these characteristics, N_1, N_2, n_1, n_2, N and n are the Halstead metrics [7] that are widely used in program similar-

Table 1: The minimum and the maximum of numerical characteristics of five sorting algorithms

Algorithm	**NoB**	**NoL**	**NoV**	**NAS**	**LoC**	**MCC**	N_1	N_2	n_1	n_2	**N**	**n**
Insertion	4/6	2/2	4/5	8/11	13/21	4/6	40/57	47/58	3/6	2/4	88/114	6/9
Selection	5/6	2/2	5/6	10/10	16/25	4/5	47/59	51/57	4/6	2/5	98/116	6/11
Bubble	5/6	2/2	4/5	8/11	15/21	4/5	46/55	49/57	4/6	2/4	95/112	6/9
Quicksort	5/9	1/3	4/7	6/15	31/41	4/10	84/112	77/98	9/17	2/7	161/198	13/22
Mergesort	7/9	2/4	6/8	14/22	33/47	6/8	96/144	94/135	11/14	5/9	190/279	17/23

ity evaluation approaches. In addition to these, some other characteristics in connection with these numerical characteristics are computed such as variable dependencies (both direct and indirect), the information whether a loop is incrementing or decrementing, and the interconnections of blocks and loops. Descriptive characteristics comprise whether the algorithm is recursive or not, whether it is in-place or requires extra memory, and the roles of variables used in it.

Based on initial manual analysis of many different versions of common sorting algorithms, we posited a hypothesis that the information mentioned above could be used to differentiate many different algorithms from each other. In the prototype version, however, we decided to restrict the scope of the work to sorting algorithms only. We studied five well-known sorting algorithms: Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort. The problem was whether new unknown code from, for example, student submissions could be identified reliably enough by comparing the information gathered from the submission with the information in a database. We developed an Analyzer that can count all these characteristics automatically. An unfortunate obstacle was, however, that the automatic role analyzer we got access to did not evaluate the roles of variables accurately enough. Due to time constraints, we could not replace it with another one, and some role analysis had to be carried out manually.

The recognition process is based on calculation of frequency of occurrence of the numerical characteristics in an algorithm on one hand, and investigation of the descriptive characteristics of that algorithm on the other hand. First, many different versions of the implementation of sorting algorithms are analyzed with regard to aforementioned characteristics and the results are stored in the database. Therefore, the Analyzer has the following information about each algorithm: the type of the algorithm, the descriptive characteristics of the algorithm and the minimum and maximum values of the numerical characteristics. When the Analyzer encounters a new submitted algorithm, it first counts its numerical characteristics and analyzes its descriptive characteristics. In the next step, the Analyzer compares this information with the corresponding information of algorithms retrieved from the database. If a match between the characteristics of the algorithm to be recognized and an algorithm from the database is found, the type of the latter algorithm is assigned to the recognized algorithm and its information is stored in the databases. If no match is found, the algorithm and its information are stored in the database as the type "Unknown". An instructor can then examine the algorithms marked "Unknown" to ensure that they really do not belong to any type of algorithms. If an algorithm marked "Unknown" does belong to a type (a false negative case), the instructor can assign the correct type to that algorithm. This way, as new kinds of implementations of an algorithm are encountered, the allowed range of numerical characteristics of that algorithm can be adjusted in the database. Thus, the knowledge base of the Analyzer can be extended: next time, the same algorithm is accepted as that particular type.

The numerical characteristics are used in the earlier stage of the decision making process to see if the recognizable algorithm is within the allowed range. If it is not, the process is terminated and the algorithm is labelled "Unknown" without any further examination. In these cases, an informative error message about the numerical characteristics that are above or below the permitted limits is given to the user. If the algorithm passes through this stage, the process proceeds to investigate its descriptive characteristics.

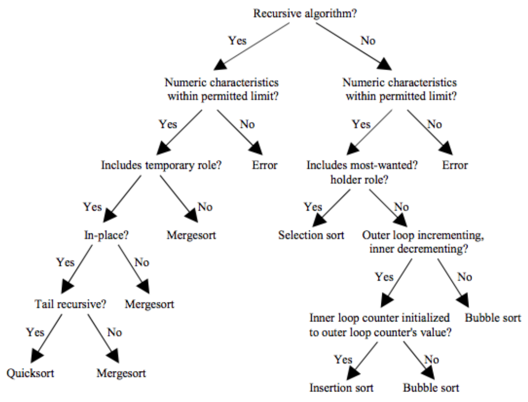

Recursive algorithm?
Yes
No
Numeric characteristics within permitted limit?
Numeric characteristics within permitted limit?
Yes
No
Yes
No
Includes temporary role?
Error
Includes most-wanted? holder role?
Error
Yes
No
Yes
No
In-place?
Mergesort
Selection sort
Outer loop incrementing, inner decrementing?
Yes
No
Yes
No
Tail recursive?
Mergesort
Inner loop counter initialized to outer loop counter's value?
Bubble sort
Yes
No
Yes
No
Quicksort
Mergesort
Insertion sort
Bubble sort

Figure 1: Decision tree for determining the type of a sorting algorithm

Figure 1 shows a decision tree to determine the type of a sorting algorithm. At the top of the decision tree, we examine whether the algorithm is a recursive one and continue the investigation based on this. Highly distinguishing characteristic like this improve the efficiency, since we do not have to retrieve the information of all algorithms from the database, but only those that are recursive or that are non-recursive. In the next step, the numerical characteristics are used to filter out algorithms that are not within the permitted limits. As can be seen from Figure 1, the roles of variables play an important and distinguishing role in the process. All examined Quicksort algorithms included a variable with Temporary role, while none of the examined Mergesorts did. Since the Temporary role often appears in swap operations, this is somehow expected: Quicksort includes a swap operation, but in Mergesort there is no need for swapping because merging is performed. In the case of the three non-recursive algorithms that we examined, only Selection sort included a Most-wanted Holder. For the definition of different roles see [20]. The rest of the decision making process shown in Figure 1 is self-explanatory.

As an example of the numerical characteristics, we present the result of analyzing the numerical characteristics of the five algorithms in Table 1. We collected an initial data base containing 51 different versions of the five sorting algorithms for the analysis. All algorithms were gathered from textbooks and course materials available on the WWW. Some of the Insertion sort and Quicksort algorithms were from authentic student submissions. For each characteristic in the table, the first and second number depict, respectively, the minimum and maximum value found from the different implementations of the corresponding algorithm. As can be seen from the table, the algorithms fall into two groups with regard to their numerical characteristics: the small group consists of Bubble sort, Insertion sort and Selection sort, and the big group comprises Quicksort and Mergesort.

4. DISCUSSION

The Analyzer is only capable of deciding which sorting algorithm a given algorithm seems to be. The correctness of the decision cannot be verified by using this method, since it is very difficult, if not impossible, to verify this using only static analysis. Dynamic methods should be used as well.

Our method assumes that algorithms are implemented using conventional and widely-accepted programming style. The method is not tolerant to the changes that result from using an algorithm in an application. Moreover, algorithms are expected to be implemented in a well-established way. As an example, although it is possible to implement Quicksort in a non-recursive way, a recursive implementation is much more common. The same assumption is made by other PC approaches as well, e.g., knowledge-base approaches.

The most useful application of the Analyzer is perhaps verifying students' submissions. There are many large size computer science courses lectured at universities where students are required to submit a number of exercises in order to complete a course. The Analyzer can be used to help instructors to verify the correctness of the type of the submissions. It is also possible to develop the Analyzer further to provide the students with detailed feedback about their submissions in different ways.

Although the method is examined only for sorting algorithms, it can presumably be applied to recognize other algorithms as well. Moreover, as we described previously, the roles of variables turn out to be a distinguishing factor that can be used to recognize sorting algorithms. This is, however, a topic well worth discussing further:

1. How well can the method be applied to recognize other algorithms?
2. What other factors could be used to characterize different algorithms?
3. Is there a minimum set of characteristics that is enough to solve the identification problem and how could it be found?
4. Roles of variables are cognitive concepts, thus a human analyzer may disagree with an automatic role analyzer. Is this causing serious problems?

5. REFERENCES

[1] 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.

[2] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceedings of the 10th European Software Engineering Conference*, pages 156–165. ACM, 2005.

[3] I. Burnstein and F. Saner. An application of fuzzy reasoning to support automated program comprehension. In *Proceedings of Seventh International Workshop on Program Comprehension, 1999.*, pages 66–73. IEEE, 1999.

[4] S. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3):1–24, 2003.

[5] B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. In *Journal of Computing Sciences in Colleges*, pages 50–57. ACM, 2007.

[6] A. Erdem, W. L. Johnson, and S. Marsella. Task oriented software understanding. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 230–239. IEEE, 1998.

[7] M. Halstead. *Elements of Software Science. North Holland, New York*. Elsevier, 1977.

[8] M. Harandi and J. Ning. Knowledge-based program analysis. *Software IEEE*, 7(4):74–81, January 1990.

[9] C. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education*, pages 46–50. ACM Press, 2002.

[10] W. Johnson and S. E. Proust: Knowledge-based program understanding. In *IEEE Transactions on Software Engineering, volume SE-11, Issue 3, March 1985*, pages 267–275. IEEE, 1984.

[11] J. Joiner, W. Tsai, X. Chen, S. Subramanian, J. Sun, and H. Gandamaneni. Data-centered program understanding. In *Proceedings of International Conference on Software Maintenance*, pages 272–281. IEEE, 1994.

[12] 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.

[13] T. J. McCabe. A complexity measure. In *IEEE Transactions on Software Engineering, volume SE-2, number 4, December 1976*, pages 308–320, 1976.

[14] D. Ourston. Program recognition. In *IEEE Expert, volume: 4, Issue: 4, Winter 1989*, pages 36–49. IEEE, 1989.

[15] A. Quilici. A memory-based approach to recognizing programming plans. In *Communications of the ACM, volume 37 , Issue 5*, pages 84–93. ACM, 1994.

[16] A. Quilici. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th international conference on Software engineering*, pages 333–336. ACM, 1995.

[17] M. J. Rees. Automatic assessment aids for Pascal programs. *SIGPLAN Notices*, 17(10):33–42, 1982.

[18] S. S. Robinson and M. L. Soffa. An instructional aid for student programs. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, pages 118–129. ACM, 1980.

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

[20] J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 37–39. IEEE Computer Society, 2002.

[21] S. Woods and Q. Yang. The program understanding problem: analysis and a heuristic approach. In *18th International Conference on Software Engineering (ICSE'96)*, pages 6–15. IEEE, 1996.

Students' Individual Differences in Using Visualizations

Prospects of Future Research on Program Visualizations

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

ABSTRACT

The range of available visualization tools for programming education is impressive but the research on them is biased mainly on testing the pedagogical effectiveness of the visualization tools. Most of the studies apply empirical techniques in controlled experimentation situations. The results on the field are summarized to be "markedly mixed".

As learning, in constructivist point of view, is seen as a process affected by the individual also the use of visualizations in learning programming depends on the learner. Instead of only studying whether visualizations in general are effective for learning, we should also study in which conditions visualizations are effective for certain kinds of learners. Controlled experimentation is also critizised as a method of studying learning since it creates artificial learning situations that do not reveal the real needs of the learner.

This article presents a literature review on the work carried out in the field of visualizations and analyzes the situation. On the basis of related work, we propose research questions for future work and discussion about research settings and methodology for achieving useful results for developing the field of visualizations further. The aim is that with this ground work we could better utilize the earlier work: visualization tools that have already been developed and the research results related to these tools.

1. INTRODUCTION

Learning programming is generally difficult. One of the big learning problems novice programmers often face is that they have to handle abstract concepts to which they do not have a concrete model in their everyday life [21]. Thus visualizations sound intuitively like a good way to concretize the abstract subject in the beginning. Actually, Stasko et al. [25] present the early development of algorithm visualizations as taking one step further from the pictural presentations that teachers use on black boards in programming classrooms.

Research on the use of visualizations in CS education has long roots too. In their book Computer Science Education Research [7], Fincher and Petre recognize animation, visualization, and simulation as one of the ten motivating areas for CS education research still in the year 2004. This reflects how much work has been put into visualization when research on teaching CS was still in its infancy. However, according to Hundhausen et al. [11], visualizations have not succeded to become a part of mainstream programming education. Among other aspects of visualizations, the reasons for this have also been studied and analyzed [19, 4]. Still, results of the research on visualizations have not been able to make a change.

The aim of this article is to deliberate how the enormous amount of work that has already been done in the field of visualizations could be utilized better to support learning programming. On the basis of visualization research and literature, the article discusses some ideas for future work on visualizations. The next section will introduce and comment the work carried out on visualization so far. Section 3 proposes some further research questions on the area and Section 4 opens a discussion on how these should be approached. Section 5 presents conclusions.

2. THEORETICAL BACKGROUND

The software visualizations (SV) used in educational purposes are often divided into two main groups according to the level on which they present the details of the software: algorithm visualizations (AV) and program visualizations (PV). AVs handle the software on a higher abstraction level. For instance, AV could present the principles of quicksort regardless of the programming language. A PV would instead present how quick sort is implemented in one specific programming language and show the implementation details of this specific program. PVs are often used in introductory programming courses when students learn programming structures using a certain programming language. AVs instead are typically used in courses concerning algorithms and datastructures. Price et al. [20] present AV and PV as two subcategories of SV but do not limit SV to these groups.

This article limits to study only the literature concerning AV and PV. Since AV is developed for much longer time than PV, most of the literature review concerns AV. For example, the most extensive articles mentioned here, a visualization research meta-study by Hundhausen et al. [11], a literature review by Stasko and Hundhausen [25], and a state of field report by Shaffer et al. [24], all concentrate mainly on research on AV. The literature review by Stasko and Hundhausen [25] does mention some PV tools too. The development of PV tools and research on them has followed similar paths than AV research. AV is so close to PV that many PV developers apply the results gained in AV research.

2.1 Development of Visualizations

By now the range of available visualization tools is impressing. There are PV tools available for the basics of practically any language used in teaching introductory programmin, for example [23], an even language independent flowchart visualizators. The range of AV tools used in the algorithm and data structure courses is even wider, for instance [17].

Many of these SV tools have been evaluated empirically to prove or measure the educational effectiveness of the tool. Still research into visualization is very tool-oriented. The evaluation studies start almost always from the tool or its features, not from the users needs.

Many of the tools are developed by expert programmers or teachers of programming which is always not only a good idea. For example, an eye-tracking study [3] reveals that expert and novice programmers use different visual attention strategies when using a visualization tool. Thus, it can be difficult for an expert to understand how the tool should be built to support the novice programmers way of using it. Stasko and Hundhausen [25] request that in the future visualization tools should be developed using a learner-centered design process and usability specialists as designers instead of CS teachers. Instead of developing tools and materials accoring to the technical visions of the developers one should study how students use visualizations and develop tools and materials according to their needs.

The learning problems in programming are often connected to more advanced issues than individual concepts, so the learning materials and visualization tools should also be directed to develop more advanced programming skills [15]. Instead of only presenting new concepts or algorithms, visualizations should also take this into account. However, most of the available visualizations tend to present concepts [24]. Research on visualizations [25, 11] requests one approach to confront this problem: the visualizations should always activate students to take part in it. Student engagement is vital for learning when using visualizations. A working group on the educational impact of visualizations has addressed this by developing a visualization engagement taxonomy that defines how intensively the learner is taking part in the visualization [18]. There are also recommendations on the pedagogical requirements of visualization tools and features that should always be implemented to a tool [22, 18]. In addition to easing the use of the visualization tool these features also give support to the learner engagement, for example, by letting the learner control the run of the visualization according to his own needs.

2.2 Studies on the Educational Effectiveness

There are plenty of studies showing that the use of visualizations makes students understand programming better, for example [5, 23, 2, 1]. On the other hand, there are some studies showing that using visualizations does not make a difference, for example [13, 9], studies where some of the experiments show a difference and some do not [8], and even a study that reports that the use of visualizations distracted the students from the essential [10].

To sum up the situation, Hundhausen et al. have performed a wide meta-study on the studies carried out on the field of AV [11]. It handles 24 different, individual studies on educational effectiveness of AV. The motivation of the meta-study is that the conclusions of the earlier studies in the field are "markedly mixed" and they want to explore deeper under the surface. They conclude that "*how* students use AV has a greater impact on effectiveness than *what* AV technology shows them." This conclusion reflects that even though a lot of work has been carried out on the field of AV and their effectiveness, the results are still vague.

The focus of the meta-study by Hundhausen et al. [11] is in the studies that use the most commonly applied method of evaluating the effectiveness of visualization, that is, AV effectiveness evaluations that employ empirical techniques in *controlled experimentation* situations. Détienne places hard methodological criticism on such studies in her psycologically-driven book of the cognitive aspects of software design [6]. *"One can [. . .] try to isolate it [a single factor in a learning situation] but at the risk of creating a rather artificial situation."* Instead of empirical research, the book promotes theoretical research for studying learning and other cognitive processes. Similarly to Detienne, Fincher and Petre emphasize that the presence of theory is important for CSER in their book [7]. Finally, the book guides researchers to carry out empirical research in CSE while taking theory into account.

In a thorough literature review of research carried out in the field of AV [25], Stasko and Hundhausen discusses similar questions about the limitations of AV effectiveness studies than the ones risen by Détienne [6] and Fincher and Petre [7]. They agree that to gain more realistic results, visualization research should in the future use other research methods than controlled experimentation. The review focuses in empirical research since theoretical research has not been carried out on AV.

In addition to controlled experiments, Stasko and Hundhausen [25] list and discuss other empirical methods that are less rigorous and have been used to research AV: *Observational studies* have been used, e.g., for researching students' understanding of visualizations and the role of visualizations. *Questionnaires and surveys* have helped to understand users' preferences, opinions, and advice regarding AV technology design and use. *Usability studies* have been used for defining the human-computer-interaction problems of AV. The literature review [25] only lists one study [10] using *ethnographic field techniques* in AV research. This study follows students constructing their own visualizations both using a visualization tool and using art supplies.

2.3 Studies on the Backgrounds of Users

One of the current learning theories, constructivism, holds that a learner constructs his own comprehension of the subject through his prior knowledge [12]. This theory emphasizes that learning is an individual process that reflects the background of the learner. Since each person learns individually also the use of visualizations in learning programmin is a personal process.

A notable remark when trawling through the research carried out on visualizations is that there is very little material on the individual differences of students when using software visualizations. Learning style, different types of programming related difficulties, and many other differences between students may still affect the way student perceives visualizations and the way he uses them.

There are studies where students are divided into a target group and a reference group randomly but later on when analyzing the results only certain kinds of students have been found to benefit from the use of visualizations. Visualizations can, for example, be beneficial only for the mediocre students [5] or the novice programmers and the students with difficulties in the programming course [1]. Also a survey on students' voluntary usage of visualizations [16] shows that the students who find the programming course too difficult or easy tend not to use program visualizations in learning. These studies show that the background of the students makes a difference. However, it is not possible to make a generalization to all visualization tools according to a few studies only.

Since the background and the personality of the student makes a difference for the use of visualizations, it could be possible that the individual differences of students are one of the explanatory factor for the "markedly mixed" results of visualization effectiveness research. Studying the background information of the students could clear the results of earlier studies. In small groups of students the differences between different kinds of students might not be statistically significant and thus this kind of aspects can be difficult to perceive and verify in many study setups.

An interesting remark is that even if the differences between students' use of visualizations have not been studied so much, there is literature on the differences of programming teachers [4]. The study devides teachers into four groups according to the way they used the visualization tool in teaching.

3. INTERESTING RESEARCH QUESTIONS

The "ultimate question" in the prior research on visualizations seems to be whether visualizations are effective in learning or not. Many of the studies address this same question with different research settings. However, this research question is on a very general level. A simple yes or no question about a phenomenon as complex as learning is close to impossible to answer. The answer naturally depends on the learner, the learning process, the visualization, the way it is used, etc., as many of the studies already recognized. This is one of the reasons why the answer to the question is still "markedly mixed" even if the question has been addressed in many studies for such a long time. Instead of seeking for an answer to this huge question, it would be beneficial to start by seeking for conditional generalizations. That is, try to find the conditions under which visualizations that have certain characteristics will be effective to learn particular things by particular students.

The conclusion of Hundhausen et al. [11] claims that the way students use AV has an important impact on effectiveness. There is only little evidence on the way students use visualizations in a real learning situation by themselves since almost all the evaluation research has been done in the artificial learning situations that Detienne [6]–for instance–critisizes. After all, most of learning takes place in students own time, in real learning situations. Thus, this path should be followed further. If we want visualizations to catch on in mainstream CS education, we need to study their usage in realistic learning situations in real CS class rooms and adapt the visualizations to suit these conditions. In addition, the research setup needs to be oriented to study the learners and not only the tool.

Shaffer et al. [24] claim that *"the theoretical foundations for creating effective visualizations are steadily improving"*. However, they demand more fundamental research on how to develop and use visualizations. To get an answer to this, we should find out who are using them and how. Only with this background information it will be possible to develop them to the right direction in the future and utilize the existing tools better. The background information will also help to gain a maximal benefit out of the existing visualization tools.

Even if the ultimate goal of research on visualizations is to find out *how visualizations should be developed* to be beneficial for students this might not be a good starting point. In order to approach the solution of this questions, we believe there is a set of more fundamental questions that needs to be answered first. For example: *What kind/type of students use visualizations in real class rooms and in real study sessions? (Real as opposed to artificial/controlled.) In what kind of situations (study sessions) do students (certain types of students) use visualizations? In which ways do students (certain types of students) use visualizations in their real study sessions?* etc.

4. AN OPENING FOR A DISCUSSION

As stated, we want to study the use of visualizations in a user-oriented manner in real learning situations.

In studies about the use of PV, there is evidence that you can create a study situation where the students choose whether they want to learn certain thing or complete a certain assignment using traditional methods (pen and paper or a normal compiler) or using a visualization tool [14]. Basically the idea in this study was to offer students the possibility to use the visualization tool in their independent study sessions but not make the use obligatory and see whom of the students use it and how. The study shows that there are different kinds of students: a group that always wants to use the visualization tool when it is possible, a group that never wants to use it, and groups of students who change their opinions about the use of the tool during the course. If the backgrounds, programming related learning difficulties, and other characteristics of these student groups could be studied wider, we could find some answers to the research questions mentioned in the end of previous section.

The contradiction in this proposal is that we wanted to approach the research problem about the use of visualizations more student-oriented and less tool-oriented. Now we propose a study setup where you offer the students a possibility to use a visualization tool upon their choise. Is that not just another variation of tool-oriented again? The problem is that it is not possible to research the use of visualization tools with no tool at all. However, instead of making a study where the students are randomly divided into a target group and a reference group and the other is forced to use the tool or its certain feature and the other one not allowed to use it, we propose to let the students make the decision upon their own interests and thus change the orientation of the study from looking at the tool or its certain features to the student and his/her background.

The proposal for topics of discussion are: Is it possible to study the use of visualization tools in a user-oriented way? Is the above mentioned proposal still too tool-oriented to gather interesting information?

5. CONCLUSIONS

The answers to the research question proposed in this article would contribute to the research on visualizations as important background information. This information could be helpful in understanding the results of earlier studies better. It could also give directions to the further development of old visualization tools and their usage. In addition, this knowledge is important if tools are in future developed using learner-centered principles as suggested [25].

The next step is to search for the answers for the discussion questions and design the research settings for the proposed research questions.

6. ACKNOWLEDGMENTS

Nokia Foundation has partly funded this work.

7. REFERENCES

[1] T. Ahoniemi and E. Lahtinen. Visualizations in Preparing for Programming Exercise Sessions. In *Proceedings of the Fourth Program Visualization Workshop*, pages 54–59, Florence, Italy, June 2006.

[2] R. Baecker. Sorting out sorting: A case study of software visualization for teachhing computer science. In *Software Visualization: Programming as a Multimedia Experience*, pages 369–381. MIT Press, 1998.

[3] R. Bednarik. *Methods to Analyze Visual Attention Strategies: Applications in the Studies of Programming.* Joensuun yliopisto, 2007.

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

[5] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

[6] F. Detienne. *Software Design – Cognitive Aspects.* Springer-Verlag, London, 2002.

[7] S. Fincher and M. Petre. *Computer Science Education Research.* Taylor and Francis, The Netherlands, Lisse, 2004.

[8] S. R. Hansen, N. H. Narayanan, and D. Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1):10, 2000.

[9] C. Hundhausen and S. Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? *Proceedings of IEEE Symposium on Visual Languages*, pages 21–28, 2000.

[10] C. D. Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach. *Computers & Education*, 39(3):237–260, 2002.

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

[12] K. Illeris. *The Three Dimensions of Learning.* Krieger Publishing Company, Malabar, Florida, 2002.

[13] D. J. Jarc, M. B. Feldman, and R. S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *SIGCSE Bull.*, 32(1):377–381, 2000.

[14] E. Lahtinen, T. Ahoniemi, and A. Salo. Effectiveness of integrating program visualization to a programming course. In *Proceedings of The Seventh Koli Calling Conference on Computer Science Education*, November 2007.

[15] 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.

[16] E. Lahtinen, H.-M. Järvinen, and S. Melakoski-Vistbacka. Targeting program visualizations. *SIGCSE Bull.*, 39(3):256–260, 2007.

[17] 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.

[18] 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.

[19] T. L. Naps, G. Rössling, J. Anderson, S. Cooper, W. Dann, R. Fleischer, B. Koldehofe, A. Korhonen, M. Kuittinen, C. Leska, L. Malmi, M. McNally, J. Rantakokko, and R. J. Ross. ITiCSE 2003 working group reports: Evaluating the educatiocal impact of visualization. *SIGCSE Bulletin*, 35:124–136, June 2003.

[20] B. Price, R. Baecker, and I. Small. An Intorduction to Software Visualizaton. In *Software Visualization: Programming as a Multimedia Experience*, pages 3–34. MIT Press, 1998.

[21] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[22] G. Rössling 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*, June 2002.

[23] J. Sajaniemi and M. Kuittinen. Visualizing roles of variables in program animation. *Information Visualization*, 3(3):137–153, May 2004.

[24] C. A. Shaffer, 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*, pages 150–154, New York, NY, USA, 2007. ACM.

[25] J. T. Stasko and C. D. Hundhausen. Algorithm Visualization. In *Computer Science Education Research*, pages 199–228, The Netherlands, Lisse, 2004. Taylor and Francis.

PatternCoder: A Programming Support Tool for Learning Binary Class Associations and Design Patterns

James H. Paterson
School of Engineering and Computing
Glasgow Caledonian University
Glasgow G4 0BA, UK
+44 141 331 3028
james.paterson@gcal.ac.uk

John Haddow
School of Computing
University of the West of Scotland
Hamilton ML3 0JB, UK
+44 1698 283100
john.haddow@uws.ac.uk

Ka Fai Cheng
School of Engineering and Computing
Glasgow Caledonian University
Glasgow G4 0BA, UK
+44 141 331 3820
k.cheng@gcal.ac.uk

ABSTRACT
PatternCoder is a software tool to aid student understanding of class associations. It has a wizard-based interface which allows students to select an appropriate binary class association or design pattern for a given problem. Java code is then generated which allows students to explore the way in which the class associations are implemented in a programming language. This paper describes the rationale behind the tool, gives a description of the tool itself, and reports on our experiences of using the tool in our teaching.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

General Terms
Design, Languages

Keywords
Java, UML, patterns, associations

1. INTRODUCTION
The transition from design to code often represents an obstacle which is difficult for students studying object-oriented design and programming to surmount. They are often taught to follow a process which includes identifying a set of classes to represent the entities in a scenario, and constructing a diagram, typically a UML class diagram, to describe these classes. The diagram will also represent the associations between the classes. UML notation can describe a number of different association types, including aggregation, generalization, and so on. The class diagram can then be a starting point for implementing the classes in a programming language such as Java to create a working solution for the original scenario.

This process presents difficulties at a number of stages. The first area of difficulty is in the design. For example, Thomasson et al. [14] identified a range of common design faults which occurred in novice designs, including a wide variation in the number of classes identified in response to a written scenario. The second area of difficulty lies in translating a completed design into code. Clearly it is unlikely that a successful implementation will result from a flawed design. However, we have observed that the implementation stage is extremely difficult for many students even when the design is valid or has been improved on the basis of feedback given.

There may be many reasons for such difficulties. However, in developing PatternCoder, we have focused on difficulties arising from a lack of a clear understanding of how classes are associated, and how instances of those classes communicate in order to perform the operations required of the system. In the study of Thomasson et al.[14], a large percentage of the student work exhibited what the authors describe as "non-referenced class faults", where students understood the need for a class to represent a concept but were unable to relate this to other classes. Sanders et al.[12] studied student understanding of object-oriented concepts. A striking observation in their results is a complete lack of recognition of the concept of message passing, and they comment that the topic of object interaction is poorly covered in many textbooks.

We suggest that placing an emphasis on teaching the specific ways in which classes and objects *can* relate and communicate with each other helps students with a key aspect of their designs as well as with implementing those designs. This has been done by having students explore a simple system designed to showcase a range of association types. The UML representation and the Java implementation were examined. This approach has similarities to the model-based approach to teaching programming described by Bennedsen and Caspersen [4] in which code patterns are used to implement specific association types. We found that the interactive Object Bench feature of the BlueJ IDE[1] is particularly useful for exploring the interactions between objects. At a more advanced level, we teach design patterns, based on the work of the so-called "gang of four" (GoF)[7], and again, there is an emphasis on the way that the classes within a pattern relate and communicate. A design pattern defines a set of classes which work together through specific relationships and collaborations. In fact, we see the fundamental class relationships essentially as simple patterns: the 'common problems' are simply the problems of modeling the ways in which two types of real-world objects

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

can be related. These relationships then form the building blocks of the more complex patterns such as the GoF patterns.

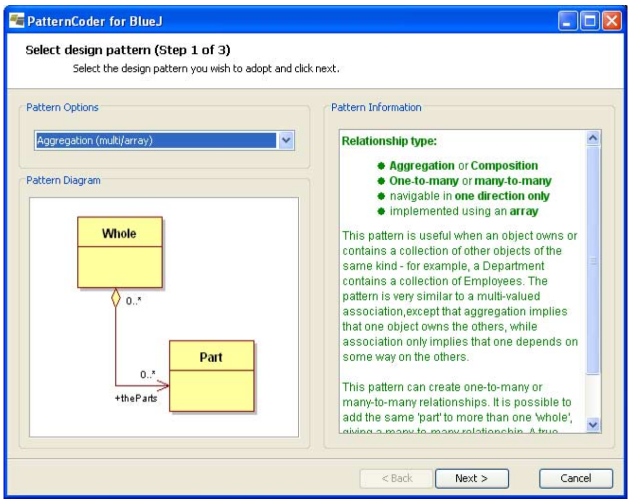
The PatternCoder tool goes a step beyond the teaching of specific examples by providing support to students as they work on their own designs and allowing them to generate their own examples.

2. DESCRIPTION OF THE PATTERNCODER TOOL

PatternCoder is essentially a code generation tool which can add a set of one or more classes to a project based on code templates. It has been developed as an extension to BlueJ. The student is initially prompted to select a pattern or association from a list, with a diagram and additional textual information given to help identify which is likely to be appropriate for the scenario which he or she is working on. The student is encouraged to think about the issues of multiplicity and navigability which will determine key implementation details. Once a decision is made, generic class names are replaced with scenario-specific names, and the classes are created within the current BlueJ project.

Knowledge of the relationships between classes is encapsulated in the code templates and in XML pattern definition files. These are text files, separate from the executable code, and can be easily customized to suit the approaches taken by different instructors. Examples of the templates and pattern files, and the way these are written, are shown in a previous paper[9].

The files supplied with the tool are designed to produce fully working code examples which allow the way the pattern or association works to be explored 'out of the box'. This is illustrated here using an example of a scenario which involves a simple ecommerce system which processes orders. The student has identified that each *order* will consist of a number of *items*, that these are modelled as Order and OrderItem classes and that these classes are associated in some way. On starting PatternCoder (which is done by selecting a BlueJ menu option), the student can explore the available patterns and their descriptions, settling in this case on a Whole-Part (or aggregation) association where the whole can contain multiple parts, as shown in figure 1.


PatternCoder for BlueJ

Select design pattern (Step 1 of 3)
Select the design pattern you wish to adopt and click next.

Pattern Options: Aggregation (multi/array)

Pattern Information

Relationship type:
- Aggregation or Composition
- One-to-many or many-to-many
- navigable in one direction only
- implemented using an array

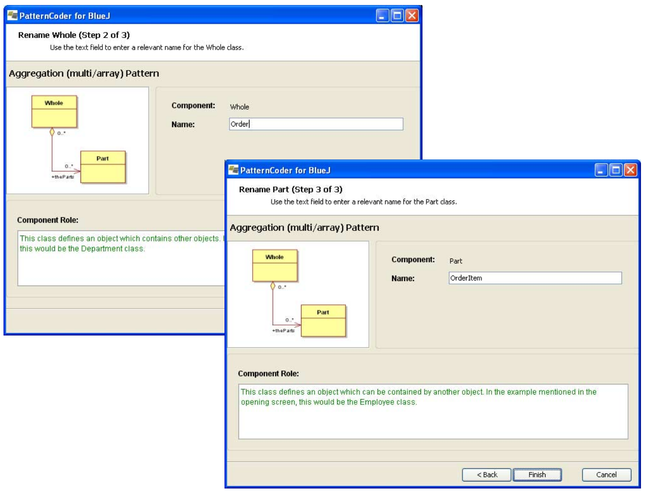
This pattern is useful when an object owns or contains a collection of other objects of the same kind - for example, a Department contains a collection of Employees. The pattern is very similar to a multi-valued association,except that aggregation implies that one object owns the others, while association only implies that one depends on some way on the others.

This pattern can create one-to-many or many-to-many relationships. It is possible to add the same 'part' to more than one 'whole', giving a many-to-many relationship. A true

Figure 1. Selecting the appropriate pattern

The following steps, generated by the XML pattern file, allow the generic names Whole and Part to be replaced with the specific names Order and OrderItem, as shown in figure 2. Finally, the Order and OrderItem classes are added to the current project. These classes do not at this point contain any scenario-specific logic, but they do contain enough code to allow the classes to be compiled, instances to be created, and generic operations such as adding and removing OrderItems to and from an Order to be performed. The student can use the generated classes in the following ways:

- as code examples to explore and understand the code required to implement the relationship. The ability to generate additional code examples may be useful to students who have difficulty in looking at examples given in lectures and extracting the necessary parts of the code to apply the given solution in another scenario.
- as a starting point to develop the complete solution by adding scenario-specific attributes and logic: for example a method to calculate the total cost of an Order. The generation of what is often referred to as "boilerplate" code has the same advantage in terms of saving time as it would for a professional developer, but for the novice has the additional benefit of allowing concentration on the logic without the frustration caused by errors or incorrect implementation of the association.


PatternCoder for BlueJ

Rename Whole (Step 2 of 3)
Use the text field to enter a relevant name for the Whole class.

Aggregation (multi/array) Pattern

Component: Whole
Name: Order

Component Role:
This class defines an object which contains other objects. I
this would be the Department class.

PatternCoder for BlueJ

Rename Part (Step 3 of 3)
Use the text field to enter a relevant name for the Part class.

Aggregation (multi/array) Pattern

Component: Part
Name: OrderItem

Component Role:
This class defines an object which can be contained by another object. In the example mentioned in the opening screen, this would be the Employee class.

Figure 2. Naming the classes

PatternCoder can be considered to simply be a conduit for transforming the knowledge encapsulated in the code templates and pattern files into working code. This means instructors are free to use their own knowledge and preferences to modify or expand on the choices and information presented to the students. The tool is supplied with a 'starter set' of design patterns which provides examples of several types of patterns but do not cover the whole catalog of GoF patterns. Instructors are free to add other patterns which they wish to teach. Similarly, a set of binary association patterns is included. These use our own nomenclature and descriptions of these associations, but instructors are able to modify them to suit the way they want to teach. Contributions from the community are welcomed and may be distributed through the *www.patterncoder.org* website. For example, there is an ongoing project which aims to translate pattern files and templates into Portuguese.

The informational text which is displayed at each step is stored in the XML pattern definition files as HTML code, allowing instructors to create content aligned with the skill levels of students as they progress through examples, and to highlight specific points in the text, and potentially to provide hyperlinks to more detailed tutorial material. This could, for example, support a set of learning activities based on the idea of scaffolding and fading out in Cognitive Apprenticeship[5].

We have recently used the customizability of PatternCoder to provide an alternative set of binary association patterns[10]. One of the difficulties in learning about these associations lies in the ambiguities caused by the gap which exists between programming languages and UML. The concept of an association does not in fact exist in Java, for example. The association must be expressed in code using the available tools: classes, attributes and methods [8]. This requires thought about exactly what is implied about the classes, and understanding of how to map that meaning to code. For example, a simple one-to-one association between two classes may be implemented using an attribute of one class. However, there may be situations where that association is temporary, and is best implemented using a reference contained in a parameter in a method call. The difference is not clearly expressed in the model, but very much affects the implementation details. These patterns, based on the work of Stevens[13] on the semantics of binary associations, are designed to encourage students to think about this kind of issue. There are many more complex aspects of class diagrams, such as qualified associations[1] which could be illustrated by creating patterns in PatternCoder, although we would certainly not wish to present these to novices.

3. RELATED TOOLS

There are many tools designed for professional developers which offer ‘round-trip’ code generation, in which changes made to a UML class diagram are immediately reflected in programming language code which is automatically generated. To take just one example, in the eUML plug-in for Eclipse[2] you can draw classes and an association between them, and edit the properties of the association. These properties are then reflected in the Java code generated for the classes. Similarly, many professional tools such as IBM Rational Developer[3] have much more sophisticated support for building design patterns into a model or project than PatternCoder offers. However, such tools are not suitable, or intended, for novice developers, are often complex, and do not offer tutorial content.

Green[2] is a round-tripping UML editor plug-in for Eclipse which is specifically designed for educational use. The aim of this tool is quite similar to that of PatternCoder, but the implementation is significantly different. Green is essentially a UML diagram editor which allows classes and associations to be added interactively to a project. PatternCoder’s wizard-driven approach, in contrast, does not try to provide any support for diagramming. Green’s knowledge of associations is encapsulated in Eclipse plug-ins, which are themselves written in Java, in contrast to the relatively easily modified templates and XML files used by PatternCoder. There is no tutorial content embedded within the tool to help students decide on the most appropriate association type. Finally, there is no specific support in Green for design patterns.

Patterns+UML[6], like PatternCoder, is an educational tool designed to provide support for the implementation of design patterns. The authors emphasize its use for exploring situations where a class can play roles in more than one design pattern, and they correctly state in comparison that PatternCoder does not support interaction between patterns. The wizard-based process for implementing a pattern appears similar to PatternCoder, Unlike PatternCoder and Green, there is no IDE integration. It is not clear in the reference whether it is possible to customize the patterns offered to the user or whether binary associations are included.

There is considerable research effort ongoing into the development of model-driven development tools which can automatically map UML associations, including the more complex types of association, into code[1,8]. This is not a trivial task, due to the conceptual gap between model and programming languages discussed in the previous section, and current professional UML tools which offer code generation support for only a limited range of associations. Genova has described a prototype code generation tool for UML associations, JUMLA [8].

Note that unlike all of these, PatternCoder does not attempt to be a model-driven development tool. It does not analyze a model and its associations in order to generate code. Instead, it asks the student to think about his or her model and to actively make a decision on what type of association should be implemented.

4. DISSEMINATION

The PatternCoder tool is open-source software, and has been made freely available for download since mid-2006. It was originally known simply as the Design Patterns extension for BlueJ. In summer 2007 the *www.patterncoder.org* website was created to distribute the tool and related materials. The download includes binaries and source code, javadocs and a guide to installing and using the tool. PatternCoder works with BlueJ on Windows, Linux and MacOS. The project code is hosted on Google Code[4] for ease of collaborative development. A basic set of teaching materials which we have developed and used are also available for download on the website. A number of papers and presentations have been given on the tool and teaching approaches based on it [9,10]. Approximately 1100 downloads were recorded over a 12 month period to June 2008. Integration with BlueJ, which is widely known and used in the CS education community, has an advantage in terms of dissemination. We are grateful to the BlueJ team for placing a link to our tool in their website, and website statistics show that this link brings a significant level of traffic to the PatternCoder site.

Pears et al.[11] noted that very few teaching tools have seen widespread adoption within CS education, and identified some possible reasons for this, including the origins of many tools as

[2] http://www.soyatec.com/euml2

[3] http://www.ibm.com/developerworks/rational

[4] http://code.google.com/p/patterncoder

solutions to local problems, and a lack of development and funding to make tools suitable for use across a wide range of institutions. A further reason may be a lack of readily available teaching materials. One of the most widely adopted tools is BlueJ, and the availability of a textbook[3] written by the tool authors and closely based on its use may be a significant factor. We propose to develop a comprehensive set of tutorial materials which will make use of and integrate with PatternCoder.

5. EVALUATION

PatternCoder has been used with our students on level 2 and level 3 modules in object-oriented programming and design. At level 2 it was introduced alongside lab exercises specifically designed to illustrate a range of class associations. Feedback from the students was very positive. Several students commented that they were now beginning to make sense of the meaning of the relationships defined in UML class diagrams and of how the relationships translate to code. It was encouraging to note from dialogue with students that many had taken the positive step of downloading PatternCoder and installing and using it on their home computers. The level 3 students found the tool helpful for revising class relationships, which many of them had struggled with previously, and some were then observed to be using the tool and applying the concepts in their projects which involved the design and implementation of a complete system.

The work submitted by the level 2 students was reviewed to identify the level of incidence of the set of common design faults reported by Thomasson et al.[14]. It should be emphasized that this is a very small scale review of the work of 20 students who were organized into groups of 3 or 4. The fault types are described as “non-referenced classes” (NRC), “references to non-existent classes” (NEC), “single attribute misrepresentation” (SAM) and “multiple attribute misrepresentation” (MAM). Details of the nature of these faults can be found in the reference. We also identified a further fault, which we refer to as “unspecified association” (UA), where an association line is drawn in the class diagram, but there is no indication at all of multiplicity, navigability or association type. The results are shown in Table 1, together with the equivalent results obtained by Thomasson et al. The striking feature apparent in the table is that *no* non-referenced class faults were observed in the work of our students, in contrast to the high incidence of this fault in the previous study. This suggests that these students at least have a clear understanding that a class must be associated with other classes in order to play a part in a system.

Table 1. Percentage of designs containing each fault

fault	this work	reference [14]*
NRC	0	89
NEC	20	31
SAM	60	50
MAM	20	15
UA	80	-

*average over 3 design exercises

Although most of the designs exhibited some examples of unspecified associations, the number of these was a small proportion of the total number of associations in each design, suggesting that there is also good understanding of the need to consider the nature of each association in order to implement it.

The results described here refer to design diagrams, whereas the specific role of PatternCoder is to support the coding of patterns and associations. However, part of the rationale of the tool is to develop understanding of the nature of associations and the need to implement them in code, and it is hoped that such understanding will feed back into improvements in design skills. It is not possible to deduce from these results the specific influence of the PatternCoder tool has had. The teaching emphasis placed on associations is likely to be a major factor. The fact that students were working in groups and discussing their designs may also have been significant. These results are based on a very small body of work and so it is not possible to draw strong conclusions. However, it does appear that a teaching approach emphasing class associations, supported by the use of PatternCoder, has the potential to improve understanding in an area which has been shown previously to cause difficulty for students.

6. LIMITATIONS AND FUTURE DEVELOPMENTS

The current release of PatternCoder has a number of issues which warrant further development. It is currently possible only to add new classes to a project, so there is no way of including an existing class within a pattern or association and modifying its code accordingly. This makes it difficult to use PatternCoder to build up a set of associations between multiple classes or to refactor a design to make use of a pattern. The authors of the Patterns+UML tool[6] emphasize this limitation in comparison to their tool, and we acknowledge and plan to address this.

Pattern file management could be improved to make it easier to install or uninstall patterns or to let the user navigate and select from within multiple named sets of patterns. Creation of pattern files and templates is done simply by editing text files (templates and XML pattern files). Although this process is documented, it would be helpful to have a tool which would provide a user-friendly interface to help instructors customize patterns and create new ones. Support for internationalization of the user interface has not yet been implemented, although the pattern files and templates can be easily edited and translated into other languages.

PatternCoder has been implemented as a BlueJ extension because we see BlueJ as a good fit for a teaching approach which emphasizes class associations. However, it would be relatively straightforward to create a standalone version, or one which integrates with other popular IDEs such as Eclipse or Netbeans.

7. CONCLUSION

Class associations and collaborations present difficulties for many students of object-oriented design and programming, something which has become apparent through our own experience and through studies reported in the literature. The PatternCoder tool was developed to support a teaching approach which emphasizes the nature and importance of associations between classes which collaborate within design patterns and through simple binary class associations. Initial experience suggests that this approach can produce significant benefits.

One of the problems with evaluating tools such as PatternCoder is that unless the tool is widely adopted, the body of student experience and work available for study is limited. It would be valuable to be able to draw together experiences with the tool from a larger number of institutions. Dissemination of the tool is a key target in order to promote wider adoption and potentially give access to a wider base for evaluation. The tool is being promoted to the community through papers and conference presentations, and a website has been designed to provide free access to the tool for instructors and students. Tutorial materials are also under development to lower the barrier to adoption of the tool. It is hoped that these efforts will drive future adoption, evaluation and development of PatternCoder.

8. REFERENCES

[1] Akehurst, D., Howells, G. and McDonald-Maier, K. (2007), "*Implementing associations: UML 2.0 to Java 5"*, Journal of Software and Systems Modelling, Vol. 6, No 1, 3–35.

[2] Alphonce, C. and Martin, B. (2005), *"Green: a customizable UML class diagram plug-in for Eclipse"*. In *Companion to the 20th annual SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, 168-169.

[3] Barnes, D.J. and Kölling, M. (2008), *"Objects First with Java. A Practical Approach"*, 4th Edition, Prentice Hall / Pearson Education.

[4] Bennedsen, J. and Caspersen, M (2008) *"Model-Driven Programming*", In *Reflections on the Teaching of Programming, Lecture Notes in Computer Science* Vol. 4821, 116-129, Springer-Verlag Berlin / Heidelberg,

[5] Collins, A., Brown, J.S. and Newman, S. (1989) "*Cognitive Apprenticeship: teaching the craft of reading, writing and mathematics"* In L. Resnick (Ed.) *Knowing, learning and instruction: essays in honor of Robert Glaser* (pp453-494). Hillsdale, NJ: Lawrence Erlbaum.

[6] Denegri, E., Frontera, G., Gavilanes, A., and Martín, P. J. (2008), *"A tool for teaching interactions between design patterns"*. In *Proceedings of the 13th Annual Conference on innovation and Technology in Computer Science Education*, 371.

[7] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *"Design Patterns: Elements of Reusable Object-oriented Software"*, Addison-Wesley, Boston, MA.

[8] Génova, G., Ruiz del Castillo, C. and Llorens, J. (2003), "*Mapping UML Associations into Java Code",* Journal of Object Technology, Vol. 2, No. 5, 135-162.

[9] Paterson, J.H. and Haddow, J. (2007), *"Tool support for implementation of object-oriented class relationships and patterns"*, ITALICS, Special Issue on Innovative Methods of Teaching Programming, Vol 6, No 4, 108.

[10] Paterson, J.H., Haddow, J and Cheng, K.F. (2008), *"Drawing the Line: Teaching the Semantics of Binary Class Associations"*, In *Proceedings of the 13th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, 362.

[11] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007), *"A survey of literature on the teaching of introductory programming"*. In *Working Group Reports on ITiCSE on innovation and Technology in Computer Science Education* (Dundee, Scotland, December 01 - 01, 2007). J. Carter and J. Amillo, Eds. ITiCSE-WGR '07. ACM, New York, NY, 204-223.

[12] Sanders, K., Bousted, J., Eckerdal, A., McCartney, R., Moström, J., Thomas, L. and Zander, C. (2008), *"Student understanding of object-oriented programming as expressed in concept maps",* In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 332-336.

[13] Stevens, P. (2002), *"On the interpretation of binary associations in the Unified Modeling Language"*, Software and Systems Modeling, Vol. 1, No. 1, 68.

[14] Thomasson, B., Ratcliffe, M. and Thomas, L. (2006), *"Identifying Novice Difficulties in Object Oriented Design",* In *Proceedings of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education*, 28-32.

9. WEBSITE

The PatternCoder website is located at *www.patterncoder.org.*

The website includes:

- Brief overview and screenshots of the tool
- Downloadable guide to installation and use
- Download of binaries, source code and Javadocs – free download, no registration required
- List of publications and other resources
- Contact information

Automatic Assessment of Program Visualization Exercises

Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso & Tapio Salakoski

Department of Information Technology
University of Turku
20014 Turku, Finland

{ertaka, temira, milaak, sala}@utu.fi

ABSTRACT

ViLLE is a visualization tool for teaching programming to novice programmers. It has an extendable support for multiple programming languages which enables language-independent learning of programming. As a new feature, ViLLE supports automatically assessed exercises. The exercises can be easily integrated into a programming course by using the TRAKLA2 web environment.

Keywords: teaching programming, novice programming, program visualization, automatic assessment.

1. INTRODUCTION

Visualization – defined as presenting a program or algorithm graphically – is a useful method in teaching novices to write and understand programs. Program visualization can be used in concretizing otherwise abstract concepts related to program execution. Typical program visualization techniques include highlighting executed code lines, color coding the program code, viewing variable information, and visualizing subprogram calls. Mere visualization, however, is usually not enough to engage students in the learning process. As Naps et al. [8] suggested, the deeper the learner's engagement with the visualization is, the better the learning results.

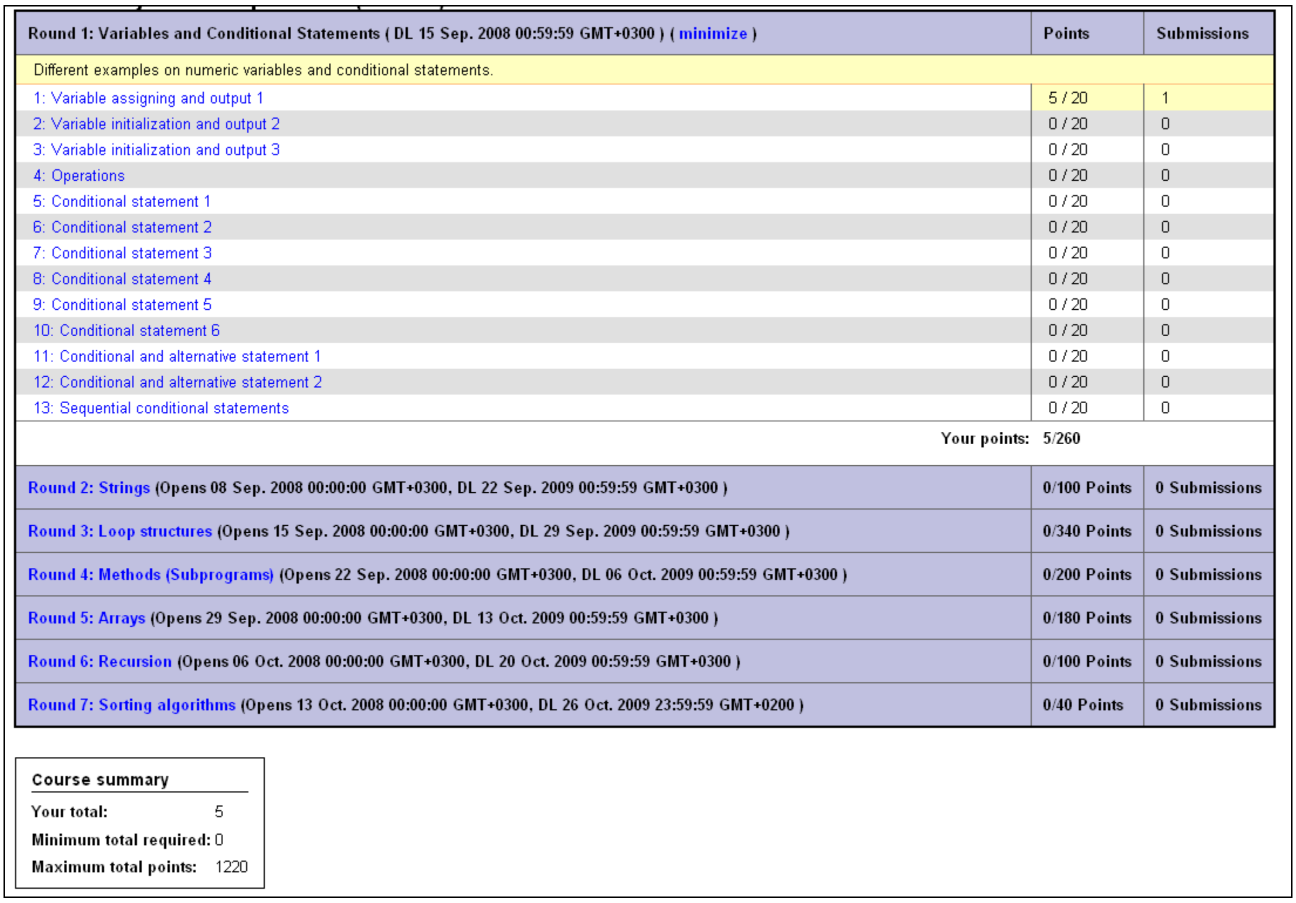
ViLLE is a program visualization tool, developed at the University of Turku. It has been designed to help the novice programmers to understand the basic concepts of program execution. One key factor in the development has been the language independency paradigm: instead of focusing on the syntax of a specific programming language, the students should understand the execution of programs in a more basic level: how do the variables work, what is a subprogram, and how the references are passed. To make this possible, ViLLE has an extendable support for programming languages; almost any language (with some limitations) can be defined with the built-in syntax editor. Moreover, the visualization of example programs can be viewed with any of the defined languages. ViLLE comes with a predefined set of languages, including Java, C++, JavaScript, Python, PHP and a pseudo language with an easy-to-follow syntax.[1]

As a new feature, ViLLE has a support for automatically assessed exercises. By using the TRAKLA2 web environment, a collection of exercises can be made available in the web. Students are awarded points on the completion of the exercises, and the teacher can easily monitor their advance. Lehtonen [5] noticed that students are more enthusiastic to use a tool, when there is a "prize" to be collected. Because the system keeps score on the completed exercises and gathered points, it's easy to include the ViLLE exercises as a mandatory part of the course.

2. RELATED WORK

Various other program and algorithm visualization tools have been developed. *Jeliot3* [7] is a program visualization tool, which focuses on teaching basic concepts of procedural and object oriented programming to novices. It relies heavily on animations, with a focus on consistency of visualizations and comprehensive support for Java features. *JIVE* [1] is a static program visualization tool, which visualizes the relationships between objects and methods. *JAVAVIS* [9] visualizes program behavior by displaying objects graphically and program execution with a sequence diagram. *BlueJ* [2] is a tool for program visualization and visual programming, allowing users to edit programs by directly manipulating their UML structure. *TRAKLA2* [6][4] is an algorithm visualization system with support for automatic assessment of exercises.

ViLLE has some unique features compared to the systems mentioned above. Firstly, the programming language support is not limited to a single language – almost any imperative language (including user-defined pseudo languages) can be added to ViLLE with the built-in syntax editor. Students can select the language any time during the execution without the need to rewrite or re-compile examples. Secondly, the controls are made flexible, including for example the ability to step backwards in the execution. In addition, all editors needed (including the syntax-, question- and example editors) are included in the package. Thirdly, the possibility to combine automatic assessment of exercises with program visualization is a feature not commonly found in similar tools.



Round 1: Variables and Conditional Statements (DL 15 Sep. 2008 00:59:59 GMT+0300) (minimize)	Points	Submissions
Different examples on numeric variables and conditional statements.		
1: Variable assigning and output 1	5 / 20	1
2: Variable initialization and output 2	0 / 20	0
3: Variable initialization and output 3	0 / 20	0
4: Operations	0 / 20	0
5: Conditional statement 1	0 / 20	0
6: Conditional statement 2	0 / 20	0
7: Conditional statement 3	0 / 20	0
8: Conditional statement 4	0 / 20	0
9: Conditional statement 5	0 / 20	0
10: Conditional statement 6	0 / 20	0
11: Conditional and alternative statement 1	0 / 20	0
12: Conditional and alternative statement 2	0 / 20	0
13: Sequential conditional statements	0 / 20	0
Your points:	5/260	
Round 2: Strings (Opens 08 Sep. 2008 00:00:00 GMT+0300, DL 22 Sep. 2009 00:59:59 GMT+0300)	0/100 Points	0 Submissions
Round 3: Loop structures (Opens 15 Sep. 2008 00:00:00 GMT+0300, DL 29 Sep. 2009 00:59:59 GMT+0300)	0/340 Points	0 Submissions
Round 4: Methods (Subprograms) (Opens 22 Sep. 2008 00:00:00 GMT+0300, DL 06 Oct. 2009 00:59:59 GMT+0300)	0/200 Points	0 Submissions
Round 5: Arrays (Opens 29 Sep. 2008 00:00:00 GMT+0300, DL 13 Oct. 2009 00:59:59 GMT+0300)	0/180 Points	0 Submissions
Round 6: Recursion (Opens 06 Oct. 2008 00:00:00 GMT+0300, DL 20 Oct. 2009 00:59:59 GMT+0300)	0/100 Points	0 Submissions
Round 7: Sorting algorithms (Opens 13 Oct. 2008 00:00:00 GMT+0300, DL 26 Oct. 2009 23:59:59 GMT+0200)	0/40 Points	0 Submissions

Course summary	
Your total:	5
Minimum total required:	0
Maximum total points:	1220

Figure 1: ViLLE exercises in the web

3. KEY FEATURES

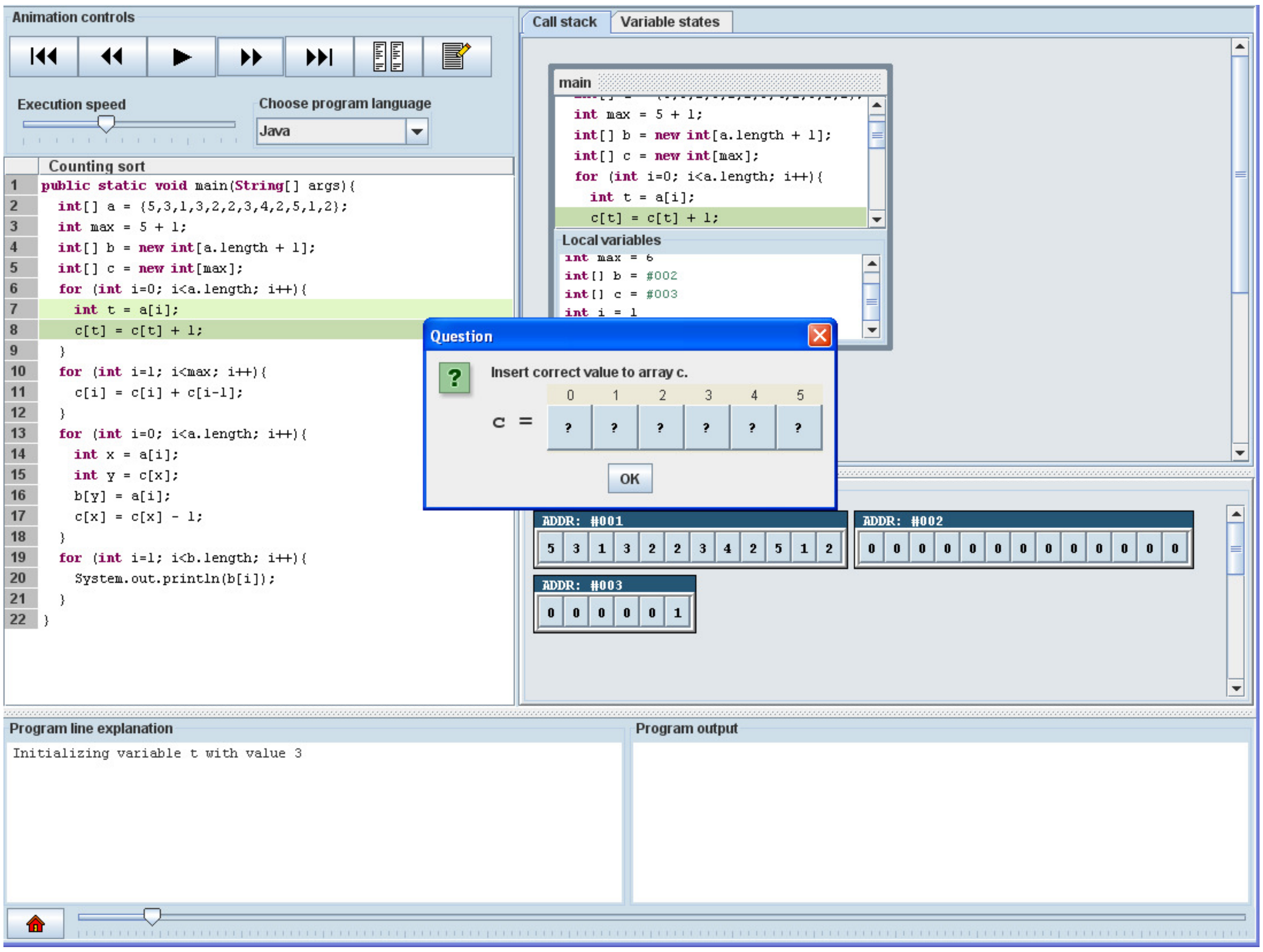
ViLLE's key features are presented here in four categories. A more comprehensive description of features can be found in Rajala et al. [10].

Level of abstraction. Some of ViLLE's features support high level of abstraction in learning to program. ViLLE has a support for multiple programming languages, including Java, Python, C++, PHP, JavaScript, and a pseudo language. Furthermore, new languages can be easily defined with the built-in syntax editor. The parallel mode in ViLLE's visualization area utilizes the support for multiple languages by visualizing the execution in two different languages simultaneously. Another abstraction of programs – the roles of variables [12] – is also supported by ViLLE.

User interaction. Teacher can create pop-questions and attach them to example programs with the built-in question editor. As a new feature, these questions can be automatically assessed (see next section). Example programs can be edited in the visualization view, which allows the user to easily see the effects of modification on program code. The animation controls in the visualization view are flexible: the user can move one step at a time both forwards and backwards or view visualization continuously with adjustable speed. There is also a slider which shows how far the visualization has progressed, and which can be used to move to any state of the program.

Execution tracing. The tool visualizes the progress of program execution by highlighting the executed code line. The progress slider has also another function: It shows the number of steps in a program, and thus it can be used in measuring the efficiency of algorithms or programs. An explanation is automatically generated for each program line. Method calls are visualized with a call stack, which shows each method call in its own frame. The progress of execution and the method's variables are displayed in the frame. ViLLE also supports the use of breakpoints.

Customization. ViLLE includes a predefined set of examples divided into different categories. These examples can be exported to a web server and thus made directly accessible to students. Teachers can also create their own example collections with the tool and use them in teaching.



Counting sort

```
public static void main(String[] args){
  int[] a = {5,3,1,3,2,2,3,4,2,5,1,2};
  int max = 5 + 1;
  int[] b = new int[a.length + 1];
  int[] c = new int[max];
  for (int i=0; i<a.length; i++){
    int t = a[i];
    c[t] = c[t] + 1;
  }
  for (int i=1; i<max; i++){
    c[i] = c[i] + c[i-1];
  }
  for (int i=0; i<a.length; i++){
    int x = a[i];
    int y = c[x];
    b[y] = a[i];
    c[x] = c[x] - 1;
  }
  for (int i=1; i<b.length; i++){
    System.out.println(b[i]);
  }
}
```

main

```
  int max = 5 + 1;
  int[] b = new int[a.length + 1];
  int[] c = new int[max];
  for (int i=0; i<a.length; i++){
    int t = a[i];
    c[t] = c[t] + 1;
```

Local variables

```
int max = 6
int[] b = #002
int[] c = #003
int i = 1
```

	0	1	2	3	4	5
c =	?	?	?	?	?	?

```
ADDR: #001
| 5 | 3 | 1 | 3 | 2 | 2 | 3 | 4 | 2 | 5 | 1 | 2 |

ADDR: #002
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ADDR: #003
| 0 | 0 | 0 | 0 | 0 | 1 |
```

```
Initializing variable t with value 3
```

Figure 2: the visualization view in ViLLE

4. AUTOMATIC ASSESSMENT

ViLLE now supports the automatic assessment of the exercises. The TRAKLA2 environment handles students' logins, and stores the points and the number of completed exercises. Teacher prepares the exercises with ViLLE's built-in question editor (currently multiple choice and graphical array questions are supported) and sends the exercises to server administrator who then prepares the course and uploads the files to the server. The example categories in ViLLE are equivalent to *rounds* in the exercise set. Each round can be given opening and closing dates. Moreover, the maximum points allowed can be defined for each example, as well as the minimum number of points required for each round.

The students can retake the exercises as many times as they want. In the future versions, the possibility to add randomized parameters to the exercises is going to be implemented. Thus, retaking the exercises becomes more challenging.

5. EVALUATION OF THE TOOL

ViLLE has been evaluated in various studies. Rajala et al. [11] evaluated the effectiveness of ViLLE at University of Turku. The students participating were randomly divided into two groups: the treatment group used ViLLE, while the control group only had access to a web based tutorial. Learning was measured with a pre- and post-test. The results showed that ViLLE was most beneficial to students with no previous programming experience, since the statistically significant difference in the pre-test between them and the experienced students disappeared after using the tool.

Laakso et al. [3] studied the effects of cognitive load in using ViLLE. While all the students gained statistically significant learning results with the system, the results showed that the students who had been familiarized with the tool beforehand, learned significantly better.

There are a few more studies with promising results not yet published. ViLLE seems to be most beneficial for novice students when used in the engagement level (see [8]) of responding. The effects of long time usage of the tool seem encouraging as well, since the students who used the system throughout entire course gained higher grades than the other students.

6. FUTURE WORK

One of the new features we are planning to include in the future versions of ViLLE is a support for exercise templates. With templates the teacher can define parts of code (or even entire code) to be randomly generated or parameterized within given limits. This way exercises can be retaken a number of times with different starting values for variables, loops or arrays. The possibility to automatically generate questions for templates or self-written examples is another feature to be implemented soon. This should make the preparation of examples more flexible and less time consuming.

More information and the system itself can be downloaded from the ViLLE homepage: http://ville.cs.utu.fi.

7. ACKNOWLEDGMENT

This work was partially supported by the Nokia Foundation and by Academy of Finland, project 121396, Automatic Assessment Technologies for Free Text and Programming Assignments.

8. REFERENCES

[1] Gestwicki, P. & Jayaraman, B. 2002. Interactive visualization of java programs, IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, 226-235.

[2] Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. 2003. The BlueJ system and its pedagogy. Computer Science Education., Special Issue of Learning and Teaching Object Technology 12(4), 249-268.

[3] Laakso, M.-J., Rajala, T., Kaila, E. & Salakoski, T. 2008. The Impact of Prior Experience in Using a Visualization Tool on Learning to Program. Appeared in Cognition and Exploratory Learning in Digital Age (CELDA 2008).

[4] Laakso, M.-J., Salakoski, T., Grandell, L., Qiu, X., Korhonen, A. & Malmi, L. 2005. Multi-perspective study of novice learners adopting the visual algorithm simulation exercise system TRAKLA2. Informatics in Education, 4(1), 49-68.

[5] Lehtonen, T. 2005. Javala – Addictive E-Learning of the Java Programming Language. In Proceedings of Kolin Kolistelut / Koli Calling – Fifth Annual Baltic Conference on Computer Science Education. Joensuu, Finland, 41-48.

[6] Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O., & Silvasti, P. 2004. Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. Informatics in Education Volume 3(2), 267-288.

[7] Moreno, A., Myller, N., Sutinen, E. & Ben-Ari, M. 2004. Visualizing Programs with Jeliot 3. In Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2004), Gallipoli (Lecce), Italy. ACM Press, New York, 373-380.

[8] Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Velázquez-Iturbide, J. Á. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. In proceeding Working group reports from ITiCSE on Innovation and Technology in Computer Science Education ITiCSE-WGR 02, 35(2), 131-152.

[9] Oechsle, R. & Schmitt, T. 2001. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In proceedings Revised Lectures on Software Visualization, International Seminar, May 20-25, 176-190.

[10] Rajala, T., Laakso, M.-J., Kaila, E. & Salakoski, T. 2007. VILLE - A language-independent program visualization tool. Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Koli National Park, Finland, November 15-18, 2007. Conferences in Research and Practice in Information Technology, Vol. 88, Australian Computer Society. Raymond Lister and Simon, Eds.

[11] Rajala, T., Laakso, M.-J., Kaila, E. & Salakoski, T. 2008. Effectiveness of Program Visualization: A Case Study with the ViLLE Tool. Journal of Information Technology Education (Innovations in Practice section), 7, 15-32.

[12] Sajaniemi J. 2002. PlanAni - A System for Visualizing Roles of Variables to Novice Programmers. University of Joensuu, Department of Computer Science, Technical Report, Series A, Report A-2002-4.

JLS/JLSCircuitTester: A Comprehensive Logic Design and Simulation Tool

David A. Poplawski
Department of Computer Science
Michigan Technological University
Houghton, Michigan
pop@mtu.edu

Zachary Kurmas
School of Computing
Grand Valley State University
Allendale, Michigan
kurmasz@gvsu.edu

ABSTRACT

`JLS` and `JLSCircuitTester` are logic design, simulation and testing tools that meet the needs of instructors and students in logic design and computer organization courses. They were designed and implemented by instructors of such courses expressly to lecture with, to do student projects, and to subsequently grade those assignments. They are free, portable, easy to install and easy to learn and use, yet powerful enough to create and test circuits ranging from simple collections of gates to complete CPUs. They come with on-line tutorials, help, and pre-made circuits taken directly from the pages of several commonly used computer organization textbooks.

1. MOTIVATION

`JLS` is a GUI-based digital logic design and simulation tool designed expressly for use by instructors and students of digital logic and computer organization. In particular, it is simple enough to be operated by an instructor during lecture, and has a very shallow learning curve necessary for student use. `JLSCircuitTester` adds functionality to support comprehensive, batch-oriented testing and grading of student-designed circuits. Together they provide an easy to use, yet powerful, mechanism to visualize the construction and operation of a wide range of digital logic circuits. Both are implemented in Java and will operate on any platform with the Java Runtime Environment (JRE) installed.

Tools like `JLS` have been around for a long time. Burch [1] and Wolffe [10] contain excellent, although slightly dated summaries many free, GUI-based circuit simulators. Many of the listed simulators are too simple to be used to construct complex circuits and CPUs because they support simple logic gates and wires only. (Some, for example, lack a mechanism to bundle wires.) Most of the listed simulators are platform specific (mainly Windows and Macintosh). A few are Java applets and, hence, have restricted functionality. (For example, applets may not load or save files.) Two are Java application programs and, hence, available on all platforms: LogicSim [9] and Logisim [1]. Another, TkGate, is implemented in C and Tcl/Tk, but can be built and installed on various platforms. Section 4 compares `JLS`/`JLSCircuitTester` with these other tools.

2. REQUIREMENTS

JLS and JLSCircuitTester have been designed and implemented by faculty who have years of practical experience teaching computer organization courses. The tools were motivated by needs for in-class presentation (lecture), by a desire to get students to test their knowledge of logic design by completing assigned projects, and the subsequent task of testing and grading those projects.

2.1 Lecture

Imagine using a digital logic simulator during lecture to demonstrate circuit concepts. For this approach to be practical, the simulator's common, fundamental operations must be quick, intuitive and easy. For example:

- Creating and placing logic elements should be convenient (drag and drop, cut and paste, etc.).
- Connecting elements (i.e., drawing wires) should require minimal work (e.g., using simple mouse clicks).
- Elements (and their connections) should be easily rearranged so that new elements can be introduced to encourage experimentation and the testing of alternatives.

In addition, instructors must be able to quickly and easily incorporate material prepared in advance and then modify it to demonstrate different principles or to help answer students' questions.

- Circuits from existing popular textbooks should be readily available so they need not be reimplmented by every instructor.
- Importing and integrating subcircuits created prior to class should require minimal effort.
- Instructors should be able to specify multiple sets of input signal values and/or sequences and memory element values in advance then easily use and modify them during lecture.

Finally, it should be easy for students to observe the step-by-step behavior of the circuit during lecture.

- Instructors should be able to choose whether the simulation of a given circuit progresses quickly to the end result, or incrementally (under instructor control) so students can observe time-varying circuit behavior.

- Signal values and memory element values should be easy to see and/or monitor in real time.

2.2 Assignments

Many instructors assign circuit design projects to their students as a way of reinforcing concepts presented in class and assessing whether the students are grasping those concepts. A good, user-friendly digital logic simulator is the ideal tool for this use.

It is usually the case that students will have little, if any, experience with logic simulation tools. Hence many of the same features that make a simulator ideal for lecture are also useful to students when preparing assignments. In addition, there are several student-oriented requirements:

- The tool should be easy to install and portable so that a student can install it on his or her platform, which may be different from the instructor's, and then use it at his or her convenience.

- Everything must be as intuitive as possible and easy to learn. (In other words, the learning curve should be very short and gradual.)

- Users should be able to use the same flexible interface to model anything from simple combinational circuits with simple gates to complex, modular circuits (such as complete CPU implementations).

- Users (especially students) must have access to tutorials that walk them through the steps of creating increasingly complex circuits using more and more features.

- On-line help must be extensive, complete and easily navigable.

- The GUI should detect and prohibit obvious errors, such as nonsensical circuits (e.g., two wires to the same input, or directly connecting two non-tristated outputs) and overlapping elements.

- Undo/redo should be available so students can easily eliminate non-working "solutions".

- Cut/paste should be supported in order to make circuit replications convenient.

- Debugging should be simple, especially with respect to slowly stepping the simulation of a circuit and the interrogation and display of signal and memory element values.

2.3 Grading

For the instructor, grading student circuits for correctness must be as convenient as possible. In its simplest terms, this means that there must be a way to load and simulate a circuit in "batch" mode (i.e., from the command line), without the GUI appearing and without further human intervention. Running a simulator in batch mode requires that input signal values and/or memory element values be configurable externally to the circuit being tested — preferably in files that are read by the simulator prior to or during simulation. To be most useful, the batch mode should have the ability to simulate the circuit under test multiple times with different, sometimes exhaustive, sets of interrelated, time varying inputs. The ability to compare circuit generated outputs with instructor-provided correct answers, and conveniently report on the results, will greatly simplify grading. In addition, providing students the ability to easily test their own circuits will improve learning by motivating students to fix problems instead of submitting an incorrect assignment and receiving a lower score [4].

3. OUR SOLUTION

`JLS` and `JLSCircuitTester` directly address the needs raised above. `JLS` is written in Java and has been tested on many platforms without modification. It consists of a simple to use, yet powerful, graphical editor that allows users to create and modify logic circuits, and a simulator that will show (in multiple ways) the operation of the circuit over a period of time. Circuits as simple as a few logic gates or as complex as complete CPU microarchitectures with embedded subcircuits have been created and simulated in JLS.

Logic circuits can contain the standard gate types: AND, OR, NOT, NAND, NOR, XOR, tri-state buffer and a logically neutral time delay element; composite elements: decoder, multiplexer, and adder; memory elements: registers, SRAM, and ROM; a clock and various mechanisms for connecting gates and elements via wires and wiring elements. Users can easily create state machines using `JLS`'s state machine editor. Combinational circuitry specified by a truth table can be generated by using `JLS`'s truth table editor. Circuits can include copies of other circuits (subcircuits), nested to an arbitrary depth. Complex, time-varying multi-signal inputs can be specified.

`JLSCircuitTester` provides a robust batch mode with which students and instructors can thoroughly test a circuit. Users provide a text file describing the set of tests to run. Each test specifies the initial input signal and memory element values as well as the expected final output signal and memory element values. `JLSCircuitTester` then simulates the circuit once for each test, compares the observed output values to the expected output values, and reports any discrepancies.

A thorough evaluation of a complex circuit requires many tests. `JLSCircuitTester` provides a number of short-cuts to simplify the generation of a large, possibly exhaustive, set of tests. For example, users can specify a list of values for each input. `JLSCircuitTester` will then generate one test for each unique combination of input values. Users can also write a Java class to calculate expected output values instead of having to type them out by hand. See [4] for more details.

4. COMPARISONS WITH OTHER TOOLS

Many existing logic design and simulation tools are very limited in the sense that they provide minimal functionality (e.g., simple logic gates and wires only). Such tools are well suited to limited situations (e.g., just simple logic) where all that is being taught matches what is available.

LogicSim has an intuitive circuit drawing mechanism and the ability to display current signal values. It also has a subcircuit (module) inclusion mechanism. However it only supports basic logic gates, flip-flops, and clocks. Wires cannot be bundled. Simple inputs values come from switch and number-input elements. Outputs are displayed by LED and LCD-like elements. There is only a limited concept of time and no concept of propagation delay. There is no batch execution mechanism and, therefore, little to assist with grading of students' circuits.

Logisim has an easy-to-use drawing mechanism and the ability to display current signal values. It supports basic logic gates, tri-state gates, flip-flops, constant value sources, and has a subcircuit inclusion. There are no other complex elements, no propagation delays and no batch execution and testing capability.

The tools most comparable to JLS in functionality are LogicWorks 5 [2] and TkGate [3]. LogicWorks is a commercial product (i.e., not free) that runs on Windows and Mac platforms only. It has little support for batch execution and grading.

TkGate is `JLS`'s principle competitor. It has features that `JLS` does not:

- A multi-lingual interface.
- Support for transistors as a basic element.
- Static critical path analysis.
- An interactive "tty" element for user interaction with the circuit.
- Customizable appearance features (e.g., colors).
- Tools for generating memory files from microcode and assembly language.
- Virtual peripheral devices (user defined input/output).
- A batch mode for which the input is a complete Verilog-based scripting language that can control many aspects of the simulator.

However, `JLS` and `JLSCircuitTester` have the following:

- Truth tables with an arbitrary number of single-bit inputs and outputs can be specified as basic elements.

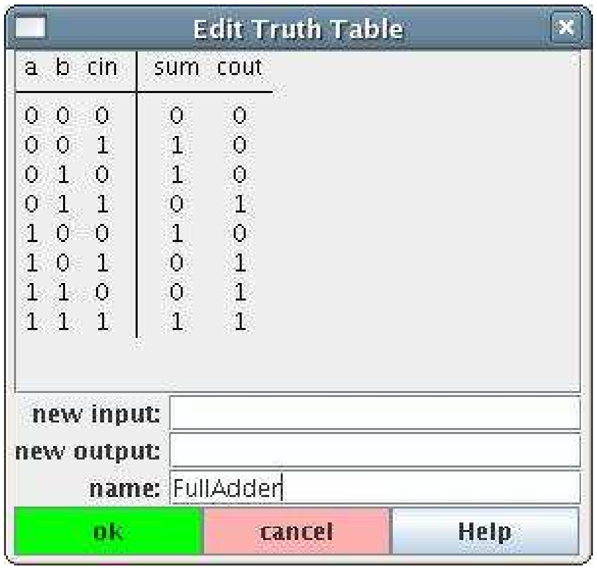
  Using a truth table can be more intuitive and convenient than a gate-level implementation of a needed but pedagogically uninteresting subcircuit.

- A state machine (Moore machine) editor.

  This feature simplifies the construction of circuits containing a state machine (e.g., CPU control) tremendously. Mapping an abstract state machine specification to hardware (state register and input/next state combinational functions), while straightforward, is tedious, error prone, and extremely difficult to modify, extend, observe in operation, and debug.

- Instructors can "lock" parts of the circuit they don't want students to modify, thereby simplifying grading because certain circuit parts will be guaranteed to be in place and correct.

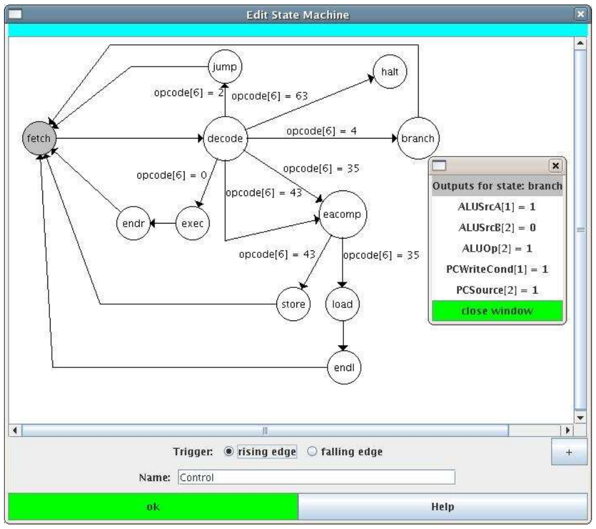
- The ability to give a wire a name, then put references to that named wire in other parts of the circuit, thereby avoiding long wire runs across the circuit that complicate the visualization.


Edit Truth Table

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

new input:
new output:
name: FullAdder

Figure 1: Truth Table Example.



Outputs for state: branch
ALUSrcA[1] = 1
ALUSrcB[2] = 0
ALUOp[2] = 1
PCWriteCond[1] = 1
PCSource[2] = 1

Trigger: [✓ rising edge] falling edge
Name: Control

Figure 2: State Machine Example.

- Complete platform independence and extreme simplicity of installation (simply download a single Java jar file for `JLS`, another for `JLSCircuitTester`).
- A library of circuits from popular computer organization textbooks is available, including directions on how to interact with the circuits to observe their behavior. Included currently are Patterson and Hennessy [6], Patt and Patel [5], and Tanenbaum [8], with more being added.
- Extensive static error checking to avoid the construction of "illegal" circuits.
- A testing framework that has a shallow learning curve and can be learned quickly regardless of experience with circuit design.

The key contributions from this list are `JLS` and `JLSCircuitTester`'s portability and ease of installation, `JLS`'s state machine and truth table elements, `JLSCircuitTester`'s powerful, yet simple, mechanisms for testing and grading circuits, and the library of existing textbook circuits.

5. PUBLICATIONS

`JLS` was first presented in a poster session at SIGCSE 2007, then in a paper presented at the 2007 Workshop on Computer Architecture Education [7]. `JLSCircuitTester` was first presented in a poster session at SIGCSE 2008. The ITiCSE paper discusses how providing students access to `JLSCircuitTester` greatly improved the quality of computer architecture projects [4] .

6. REFERENCES

[1] Carl Burch. Logisim: a graphical system for logic circuit design and simulation. *J. Educ. Resour. Comput.*, 2(1):5–16, 2002.

[2] Capilano Computing. *LogicWorks 5 Interractive Software*. Prentice Hall, 2003.

[3] Jeffrey Hansen, 2006. http://www.tkgate.org.

[4] Zachary Kurmas. Improving student performance using automated testing of simulated digital logic circuits. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 265–270, New York, NY, USA, 2008. ACM.

[5] Yale Patt and Sanjay Patel. *Introduction to Computer Systems: From Bits & Gates to C and Beyond*. McGraw Hill, second edition, 2004.

[6] David Patterson and John Hennessy. *Computer Organization and Deisgn: The Hardware/Software Interface*. Morgan Kaufmann, third edition, 2005.

[7] David A. Poplawski. A pedagogically targeted logic design and simulation tool. In *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education*, pages 1–7. ACM, 2007.

[8] Andrew Tanenbaum. *Structured Computer Organization*. Prentice Hall, fifth edition, 2006.

[9] Andreas Tetzl, 2006. http://www.tetzl.de/java_logic_simulator.html.

[10] Gregory S. Wolffe, William Yurcik, Hugh Osborne, and Mark A. Holliday. Teaching computer organization/architecture with limited resources using simulators. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 176–180, New York, NY, USA, 2002. ACM Press.

PeerWise

Paul Denny, John Hamer, and
Andrew Luxton-Reilly
Dept. of Computer Science
University of Auckland
Auckland, New Zealand
{paul, j.hamer, andrew}@cs.auckland.ac.nz

Helen Purchase
Dept. of Computing Science
University of Glasgow
Glasgow, United Kingdom
hcp@dcs.gla.ac.uk

ABSTRACT

PeerWise is a web-based system that allows multiple-choice question banks to be built solely from student input. The system provides a number of intrinsic reward structures that encourage students to contribute high-quality questions in the complete absence of instructor moderation. Several opportunities for learning arise, spanning the range from simple drill-and-practice exercises to deep, reflective study. Affective skills are also developed, as students are challenged to give and receive critical feedback and provide quality judgements.

The system is freely available, and has been used in a range of disciplines in two Universities.

Categories and Subject Descriptors: K.3.1 Computers and Education: Computer Uses in Education

General Terms: Human factors.

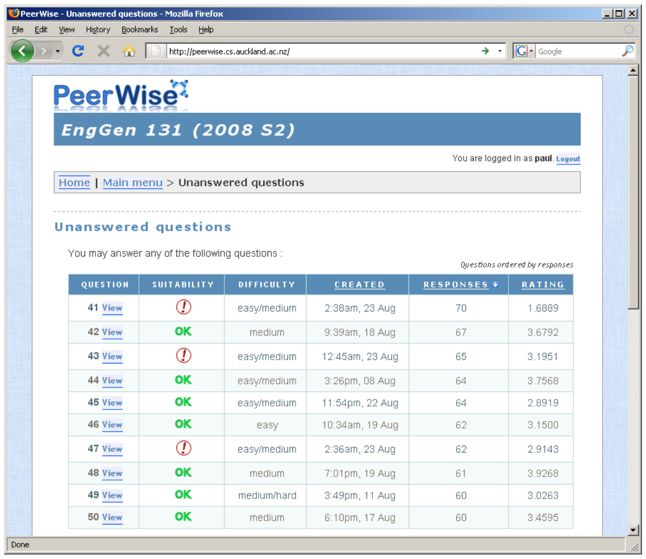
Keywords: MCQ, peer assessment, automated, question test bank, PeerWise, contributing student.

1. RATIONALE

There is a lot to learn in computing, not least in introductory programming. Some of it is quite subtle (such as integer-float widening rules, or operator precedence), some of it can be encyclopedic in nature (API libraries), and much requires high-level cognition (problem decomposition, algorithm design, coding).

Further, as teachers in a professional discipline, we have a responsibility to prepare students with the skills necessary for them to function effectively in the knowledge economy. These skills include: working collaboratively, making informed judgements, presenting information, etc. [3, 4].

PeerWise addresses these challenges by allowing students to compose multiple-choice questions and contribute them to a shared repository, where they are available for answering by other members of the class. Student identities are kept confidential; i.e. although PeerWise knows who contributed and answered each question, this information is not revealed to users. Figure 1 shows the main selection screen that students use to select the questions they answer.


PeerWise - Unanswered questions - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://peerwise.cs.auckland.ac.nz/
Google
PeerWise
Logout
Home
Main menu
Done

EngGen 131 (2008 S2)

You are logged in as **paul**.

Home | Main menu > Unanswered questions

Unanswered questions

You may answer any of the following questions :

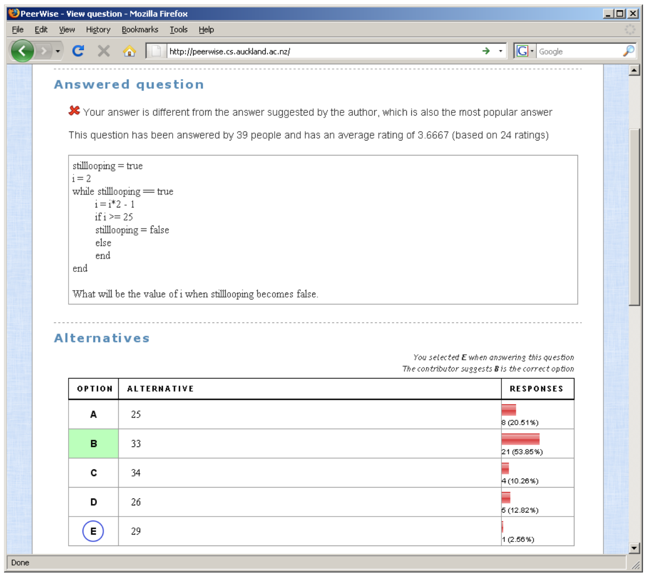
Questions ordered by responses

QUESTION	SUITABILITY	DIFFICULTY	CREATED	RESPONSES	RATING
41 View	!	easy/medium	2:38am, 23 Aug	70	1.6889
42 View	OK	medium	9:39am, 18 Aug	67	3.6792
43 View	!	easy/medium	12:45am, 23 Aug	65	3.1951
44 View	OK	easy/medium	3:26pm, 08 Aug	64	3.7568
45 View	OK	easy/medium	11:54pm, 22 Aug	64	2.8919
46 View	OK	easy	10:34am, 19 Aug	62	3.1500
47 View	!	easy/medium	2:36am, 23 Aug	62	2.9143
48 View	OK	medium	7:01pm, 19 Aug	61	3.9268
49 View	OK	medium/hard	3:49pm, 11 Aug	60	3.0263
50 View	OK	medium	6:10pm, 17 Aug	60	3.4595

Figure 1: Students have access to all contributed questions.

Several opportunities for learning arise in this process. First, the student must choose a topic and understand it deeply enough to be able to frame a suitable question. The question will often be inspired from a misconception or difficulty with which the student struggled when learning the topic. The question must include a stem, a set of alternatives, an indication of which alternative is correct, and a written explanation for why this alternative is correct and the other alternatives are wrong. Questions can also be tagged with keywords, in which case the student needs to think about how the question relates to the course.

A second phase of learning arises when students select and answer questions from the repository. If their answer differs from the alternative deemed correct by the author, they need to reflect on who is in error. A frequency table of responses from other students is displayed, so students know how many other students gave the same or different answer. Figure 2 shows the student view of the frequency table.


PeerWise - View question - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://peerwise.cs.auckland.ac.nz/
Google
Done

Answered question

✗ Your answer is different from the answer suggested by the author, which is also the most popular answer

This question has been answered by 39 people and has an average rating of 3.6667 (based on 24 ratings)

```
stilllooping = true
i = 2
while stilllooping == true
      i = i*2 - 1
      if i >= 25
      stilllooping = false
      else
      end
end

What will be the value of i when stilllooping becomes false.
```

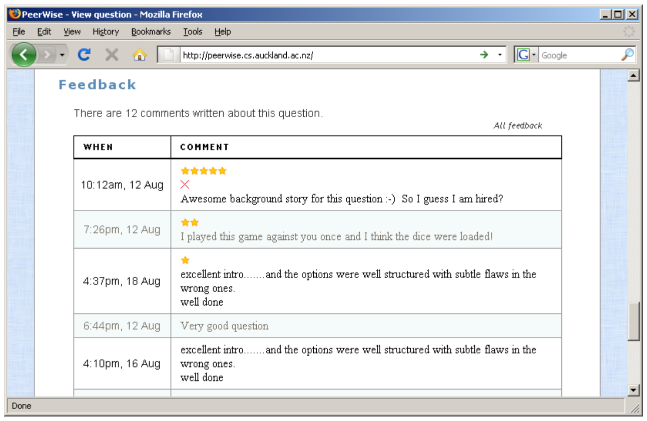
Alternatives

*You selected **E** when answering this question*
*The contributor suggests **B** is the correct option*

OPTION	ALTERNATIVE	RESPONSES
A	25	8 (20.51%)
B	33	21 (53.85%)
C	34	4 (10.26%)
D	26	5 (12.82%)
E	29	1 (2.56%)

Figure 2: Students can see how their classmates responded to the question.

Students are encouraged to apply critical analysis skills by judging the contributions of others. They are able to rate both the quality and the difficulty of a question, as a means of providing feedback to the author and to encourage or discourage other students from attempting it. PeerWise provides a discussion thread on each question, to allow uncertainties to be debated and for conveying possible improvements. Students are able to express their agreement or disagreement, displayed as a small star or cross, with any comment posted and the comments are displayed in the discussion thread in order of agreement. Prolific authors and authors who contribute popular or highly rated questions are shown on leader boards, which serve to stimulate high quality engagement with the system. Figure 3 shows a typical discussion thread associated with a question.


PeerWise - View question - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://peerwise.cs.auckland.ac.nz/
Google
Done

Feedback

There are 12 comments written about this question.

All feedback

WHEN	COMMENT
10:12am, 12 Aug	★★★★★ ✗ Awesome background story for this question :-) So I guess I am hired?
7:26pm, 12 Aug	★★ I played this game against you once and I think the dice were loaded!
4:37pm, 18 Aug	★ excellent intro.......and the options were well structured with subtle flaws in the wrong ones. well done
6:44pm, 12 Aug	Very good question
4:10pm, 16 Aug	excellent intro.......and the options were well structured with subtle flaws in the wrong ones. well done

Figure 3: A discussion thread is included with each question.

PeerWise was first used in 2006, and is currently used by courses in Computer Science, Engineering, Population Health and Pharmacology at The University of Auckland, and Computer Science and Chemistry at The University of British Columbia. The classes range in size from 16 to 869.

1.1 Typical usage

PeerWise works best with large classes, ideally with a hundred or more students. We have limited experience in its use with small classes. Our best practice recommendations are to require each student to contribute a small number of questions (perhaps two) and to answer, say, ten or twenty questions. Awarding a few marks for achieving these minimal participation requirements is usually sufficient to ensure a rich question bank.

We have not found it necessary to restrict the choice of topic, or to insist on students posting comments. Some guidance in selecting keyword tags may be appropriate, however.

Generally, students are given several weeks in which to contribute their questions, followed by a shorter period in which to answer the minimum requirement. There is usually no need to close access to the system until after the end of the course, as students are likely to voluntarily use the repository as a revision tool.

2. EVIDENCE FOR SUCCESS

We have looked at the following measures of success:

2.1 Student usage patterns

Usage log data provides detailed information of when students are using PeerWise and what they are doing. We have found distinct patterns of use for contributing and for answering questions [6]. Where contributions of new questions are required by a specific date, there is a distinct peak leading up until that date, with little or no activity after. Students tend not to continue writing new questions after the assessment of that component is complete. Most students contribute the minimum number of questions.

Answering questions follows a different pattern. There is a distinct peak before the minimum answer due date, but further peaks precede test and examination dates. However, most feedback on the quality of questions is done during the time questions are being contributed.

These patterns are consistent across different courses, and do not appear to be dependent on the lecturer, grading incentives, or use of MCQ questions in the final exam. They show that students value PeerWise as a revision tool, and are willing to spend time using it without any explicit incentive.

2.2 Examination performance

We have also done a correlation study between PeerWise activity and overall course performance [5]. Our approach involved dividing the students into quartiles based on their performance in a mid-semester test that was administered before any use of PeerWise. Each quartile was then divided into equal-sized "most PeerWise active" and "least PeerWise active" groups, using various measures of activity (number of contributed questions, number of questions answered, number of comments, total size of comments, number of days active, and a combined measure).

We found a significant correlation between all activity measures and performance on the MCQ section of the exam. Further, three of the activity measures (total length of comments, days active, and the combined measure) showed a significant correlation with the written examination questions. There is no reason to expect that extensive experience with MCQs would help in performance of written questions, unless such experience led to a deeper understanding of the material. Our results suggest it is not merely the activities of creating and answering MCQs that result in improved non-MCQ performance, but a high engagement with PeerWise (as evidenced by comments and activity days). This engagement thus suggests the development of a deeper level of understanding.

2.3 Course coverage

We have looked at the topics on which students choose to write questions and the "tags" they use to classify their questions, in a course that provided neither guidance or incentive to influence student choice. Our interest in this study was in the coverage of PeerWise question banks, and in the ability of students to come up with accurate tag.

We found that the coverage was indeed comprehensive, and the tagging was largely effective.

2.4 Question quality

We studied the quality of questions created by students in a large, first-year programming course [7]. We found that students are capable of writing questions that faculty judge to be of high quality. The best questions have well written question stems, good distracters and detailed explanations that discuss possible misconceptions.

We inspected a sample of questions closely to study specific aspects of question construction. In particular, we assessed the clarity of the language used describing the question and recorded whether any minor grammatical errors were present. We also inspected the set of distracters for each question, recording how many of them were meaningful and feasible. Finally, we classified the usefulness of the explanations written by question authors.

Because students can write comments about the questions they answer, even poorly written questions can become useful learning resources if the comments left by others are insightful. In all of the cases we examined, questions with incorrect solutions were identified by other students in the class, and comments describing the mistake were provided.

We also looked at the accuracy of the student ratings (i.e. how accurately the students were able to judge the quality of the questions). The judgements that students make about the quality of questions they answer are reasonably accurate, and correlate strongly with staff judgements. This is particularly interesting since students are using questions as a learning resource and a self-assessment tool, whereas staff generally use questions for summative assessment and diagnosis of misconceptions.

These results suggest that not only can students create high quality questions, but their ability to accurately determine the quality of the questions created by other students ensures that high quality questions are answered more frequently than low quality questions. Figure 4 illustrates how the high quality questions in a typical course are answered more frequently than the low-quality questions.

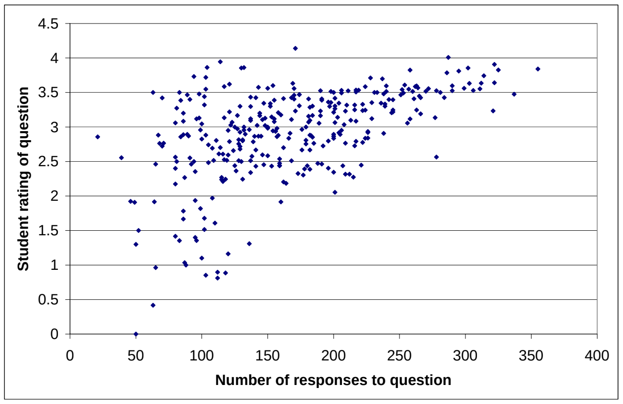


Figure 4: Students choose to answer high quality questions more frequently than low quality questions

3. RELATED WORK

The idea of student-contributed questions is not new, and a number of initiatives that share our aims have been reported in the literature.

Horgen [9] used a lecture management system to share student generated MCQs. Fellenz [8] reported on a course where students generated MCQs which were reviewed by their peers, although technology was not used to support this process. Fellenz reported that the activity increased student ownership of the material and motivated students to participate. Barak [2] reports on a system named QSIA used in a postgraduate MBA course in which students contribute questions to an on-line repository and rank the contributions of their peers. Arthur [1] reports on a large course activity in which students in one lecture stream prepare questions for a short quiz which is then presented to students in another stream. The questions are stored in a simple electronic repository. Yu [10] has students construct MCQ items and submit them to an on-line database where they are peer-assessed. Feedback about quality is used to improve the items before they are transferred to a test bank database to be used for drill-and-practice exercises.

All of these reports agree that student-contributed MCQs is a powerful idea. The major contribution of PeerWise is a tool that is simple for new institutions to adopt and which supports very large classes with little or no moderation required by instructors.

4. REFERENCES

[1] N. Arthur. Using student-generated assessment items to enhance teamwork, feedback and the learning process. *Synergy*, 24:21–23, Nov. 2006. `www.itl.usyd.edu.au/synergy`.

[2] M. Barak and S. Rafaeli. On-line question-posing and peer-assessment as means for web-based knowledge sharing in learning. *International Journal of Human-Computer Studies*, 61:84–103, 2004.

[3] M. Birenbaum. Assessment 2000: toward a pluralistic approach to assessment. In M. Birenbaum and F. Dochy, editors, *Alternatives in Assessment of Achievement, Learning Processes and Prior Knowledge*, pages 3–31, Boston, MA., 1996. Kluwer Academic.

[4] B. Collis. The contributing student: A blend of pedagogy and technology. In *EDUCAUSE Australasia*, Auckland, New Zealand, Apr. 2005.

[5] P. Denny, J. Hamer, A. Luxton-Reilly, and H. Purchase. Peerwise: students sharing their multiple

choice questions. In *ICER'08: Proceedings of the 2008 International Workshop on Computing Education Research*, Sydney, Australia, Sept. 2008.

[6] P. Denny, A. Luxton-Reilly, and J. Hamer. The PeerWise system of student contributed assessment questions. In Simon and M. Hamilton, editors, *Tenth Australasian Computing Education Conference (ACE 2008)*, volume 78 of *CRPIT*, pages 69–74, Wollongong, NSW, Australia, 2008. ACS.

[7] P. Denny, A. Luxton-Reilly, and B. Simon. Quality of student contributed questions using peerwise. In M. Hamilton and T. Clear, editors, *ACE'09: Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, CRPIT, Wellington, New Zealand, Jan. 2009. ACS. (submitted for publication).

[8] M. Fellenz. Using assessment to support higher level learning: the multiple choice item development assignment. *Assessment and Evaluation in Higher Education*, 29(6):703–719, 2004.

[9] S. Horgen. Pedagogical use of multiple choice tests - students create their own tests. In P. Kefalas, A. Sotiriadou, G. Davies, and A. McGettrick, editors, *Proceedings of the Informatics Education Europe II Conference*. SEERC, 2007.

[10] F.-Y. Yu, Y.-H. Liu, and T.-W. Chan. A web-based learning system for question posing and peer assessment. *Innovations in Education and Teaching International*, 42(4):337–348, Nov. 2005.

Towards Students' Motivation and Interest – Teaching Tips for Applying Creativity

Ralf Romeike
Department of Computer Science
University of Potsdam
A.-Bebel-Str. 89
14482 Potsdam, Germany
romeike@cs.uni-potsdam.de

ABSTRACT
Our research revealed creativity as a pathway to computer science in the biographies of CS freshman. Furthermore the application of creativity in CS classes was found to be a powerful instrument to address students' motivation and interest. This poster concludes the findings of the research projects by giving concrete teaching tips of how creativity could be regarded when planning and conducting CS lessons.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education, Literacy, Self-assessment.*

General Terms
Experimentation, Human Factors.

Keywords
Creativity, Computer Science, Wider Access, Computers and Society, CSE Research, Pedagogy, Motivation.

1. INTRODUCTION
Creativity research suggests that the application of creativity in education has the potential to raise students' motivation and interest and thus their achievement.

In a study investigating the question whether this potential is reflected in biographies of CS majors creativity was found to form a potential pathway to CS [1]. Students in whose pathways creativity characteristics were found perceived CS as fun, creative, and self-exploring. It was found that it was most rewarding for students to strive for good working software, based on self chosen and meaningful tasks, even if the resulting products did not have any outside value. Most important to the students were their activities (mostly programming), what is typical for creative processes.

The application of creativity in CS high school classes approved the potential when creativity is applied – motivation, fun, interest and achievements improved in a high school programming lecture [2]. The implications of this research for applying creativity in teaching are illustrated in Fig. 1 and summarized as follows.

2. APPLYING CREATIVITY
According to our research the use and application of creativity is based on three drivers: the subject and habits, the regard of personal factors and issues and the choice of creativity supporting tools, which are mainly responsible for a creative environment [3]. The creativity supporting factors are interrelated: interest stimulates motivation and vice versa; especially design processes are interesting to students. Creativity in CS especially gets obvious in (software) design processes, which generally culminate in a product. In CS lessons such a product is the goal of a constructive learning process and results from the adequate composition of CS concepts represented as building blocks. The CS concepts are fundamental for an understanding and efficient use of ICT and the potential for constructing artifacts with ICT, which again fosters students' motivation to engage in computing.

According to these drivers the implications are illustrated in Fig. 1 and summarized as follows. Further explanations and references to the underlying research can be found in the papers cited.

2.1 Computer Science as a creative subject and activity
In CS creativity can be fostered due to the characteristics of the subject. This especially gets obvious to students in programming and should be supported by applying tasks that reflect the creative sides of programming. This can be done by

- focusing on the products as well as on the creative process
- allowing tinkering around and trial & error for finding a solution
- applying a building block metaphor when introducing CS concepts

2.2 Regarding personal factors
Creativity requires intrinsic motivation; also, intrinsic motivation can be stimulated by creative activities. In CS lessons creative activities stimulate motivation and interest of the students. This is more likely to occur if the tasks assigned

- are meaningful to students

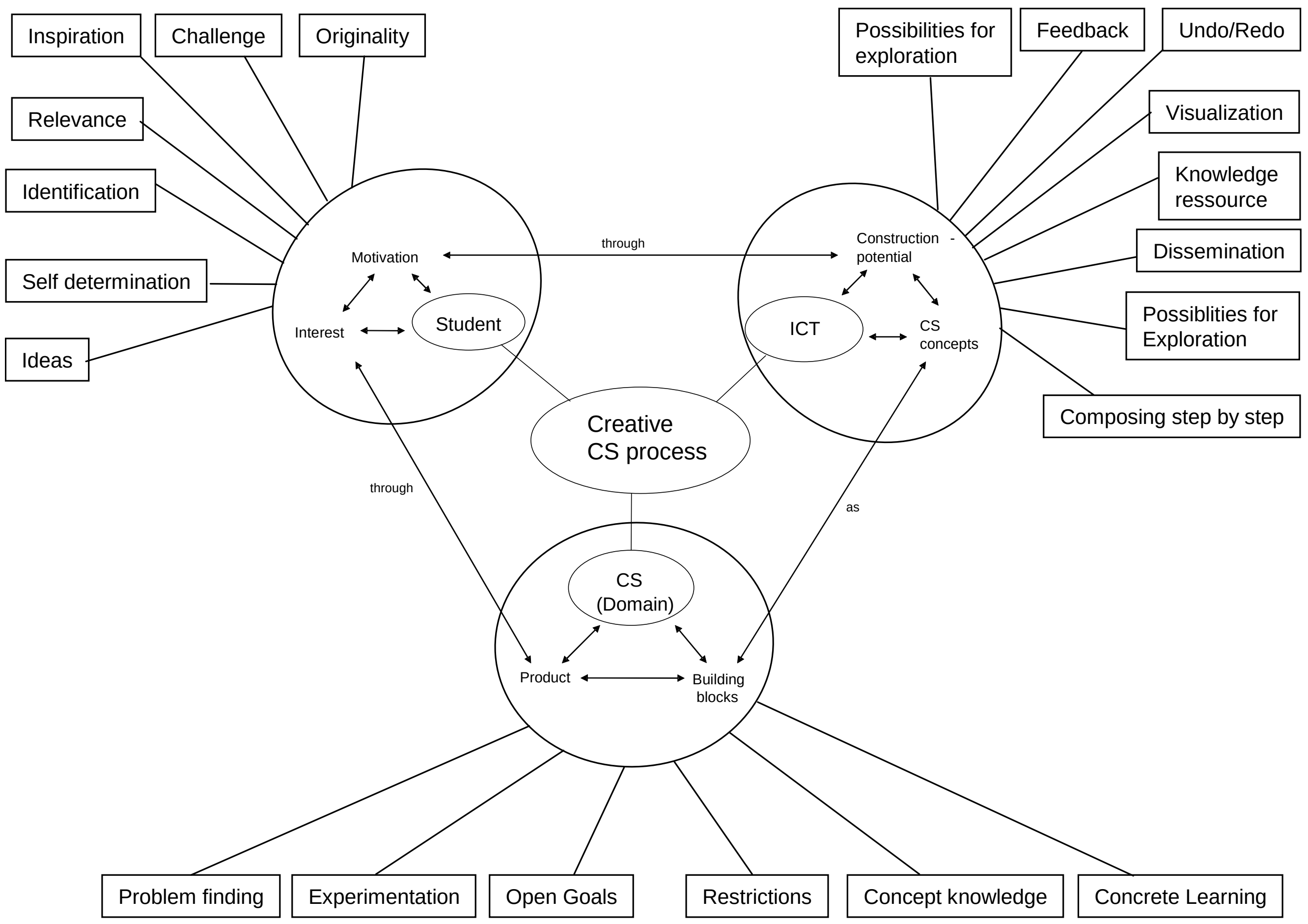


Fig. 1: Factors and implications for creative computer science lessons.

- are open tasks and thus allow for problem finding and problem defining
- are contextualized tasks

2.3 Choosing ICT that support creativity

Information- and Communication Technologies (ICT) have the potential to foster creativity in CS. Many of the tools available for teaching in CS per se fulfill creativity characteristics. Also, ICT form a major part of the learning environment. In order to ensure best possibilities for creativity tools need to be chosen that support [4]

- pain-free exploration and experimentation
- immediate and useful feedback for one's actions
- no big penalties for mistakes, meaningful reward for success
- easy way to undo and redo
- visualizing data processes
- searching for knowledge and inspiration
- composing a work step by step
- disseminating results to gain recognition

3. CONCLUSION

The creative perception of CS is a very attractive one. We highly encourage educators to try out the possibilities of creativity and the chances it offers for teaching CS.

4. REFERENCES

[1] Knobelsdorf, M. and Romeike, R. Creativity as a Pathway to Computer Science. In Proc. of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITICSE 2008), Madrid. ACM Press, 2008.

[2] Romeike, R. Applying Creativity in CS High School Education - Criteria, Teaching Example and Evaluation. In Proc. of the 7th Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Koli, Finland, 2008.

[3] Romeike, R. Three Drivers for Creativity in Computer Science Education. In Proc. of the IFIP-Conference on "Informatics, Mathematics and ICT: a golden triangle", Boston, 2007.

[4] Shneiderman, B. Creativity support tools. Commun. ACM, 45 (10), 2002, 116-120.

Presentation of Automatic Conflictive Animations

Andrés Moreno
Department of Computer Science and Statistics
University of Joensuu, Finland
firstname.lastname@cs.
joensuu.fi

Niko Myller
Department of Computer Science and Statistics
University of Joensuu, Finland
firstname.lastname@cs.
joensuu.fi

General Terms

Human Factors, Languages

Keywords

CS1, animation, programming, conflictive animation, constructivism

1. INTRODUCTION

Conflictive animations is an approach to use animations in programming education which was introduced at last year's Koli Calling [4]. Conflictive animations are created so that they do not animate faithfully what the programs intend to do. They aim to compel the student to critically review the animation by asking them to spot possible errors or mistakes in the animation. Thus, students take a new role in their relation to educational tools, which are now prone to fail.

Previously, program visualization has been used to demonstrate the fundamental concepts of programming. Lecturers present programming concepts along its visual representation using program visualization tools. Programs written by teachers or students are animated step by step showing how the computer executes its statements. Teachers can concentrate in the explanations as the tool provides the correct graphical representation. Students can later use these same tools to review the lessons or debug their own programs.

Automatic generation of program visualizations is possible with tools like Jeliot 3 [3]. Algorithm animation tools like MatrixPro [2] allow to create animations of certain data structures and operations on them automatically. These tools solve one of the common issues teachers have with algorithm animation; they often complain about the time consuming task of creating animations of their own examples [5].

To allow for a convenient way of creating conflictive animation activities we have started implementing them in Jeliot 3, which is described in Section 2. Jeliot 3's architecture has been modified to allow the development of conflictive animations for individual concepts. These modifications and new user interaction features are shortly discussed in Section 3.

2. JELIOT 3

In order to demonstrate the capabilities of conflictive animations in programming learning, we have used Jeliot 3, a program visualization environment. Jeliot has been effective in improving the learning of elementary computer science and programming [1]. It visualizes Java programs automatically without any user involvement. Thus, novices can start using Jeliot 3 on their first day of learning to program. Jeliot 3 supports the addition of stop and think questions, which ask the students about the result of the following statement or expression.

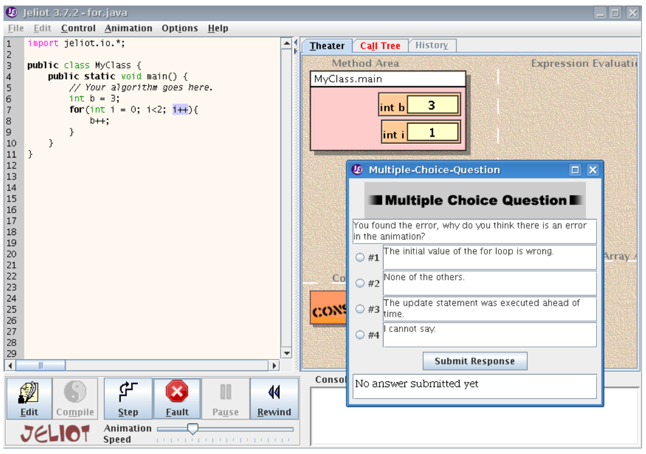
Fundamental components of the GUI can be seen in Fig. 1. The source code editor is in the left-hand pane, while the right-hand pane is used to display the visualization. VCR-like buttons to control the visualization are located in the lower left corner. Fully dynamic animation of the data and control flow of the program is displayed, including every aspect of the program execution (e.g, method calls, object construction, and expression evaluation). The animation is created automatically from the source code, so that the student needs only to learn to use the control buttons in order to work with Jeliot 3.

3. CONFLICTIVE FEATURES

Jeliot 3's modular architecture has been extended to add support for alternate and potentially conflictive animations. Jeliot 3's Java interpreter has been altered to produce a faulty interpretation of correct source code. The Java interpreter notifies the start and end of a conflictive animation to the visualization engine using an special instruction. This information is used by the user interface to support, i.e., give feedback, the student finding the error.

Three concepts are already implemented in Jeliot 3 for potential conflictive animation including basic loop operation and advanced inheritance concepts. Thus, any source code that contains the implemented *conflictive concepts* leads to a conflictive animation that students can interact with. The framework that has been implemented in Jeliot 3 provides the structure to add more conflicts for other concepts with relative ease.

New potential conflicts could reflect common misconceptions of students' understanding of Java execution, but it is more important the new role the student takes when watching the animation than the conflict itself.



```java
import jeliot.io.*;

public class MyClass {
    public static void main() {
        // Your algorithm goes here.
        int b = 3;
        for(int i = 0; i<2; i++){
            b++;
        }
    }
}
```

Multiple-Choice-Question

Multiple Choice Question

You found the error, why do you think there is an error in the animation?

- #1 The initial value of the for loop is wrong.
- #2 None of the others.
- #3 The update statement was executed ahead of time.
- #4 I cannot say.

Submit Response

No answer submitted yet

Figure 1: Conflictive user interface of Jeliot 3. In the picture, the student has pressed the Fault button and asked why she thinks there is an error.

3.1 User Interaction

Jeliot 3 UI needs to change when users are dealing with conflictive animations. As the mission of the student is to spot the error, we include a Fault button that, when pressed, indicates to the visualization tool that the user thinks a conflict (i.e. error) has occurred. Moreover, the Play button has been removed to force the student go step by step. Figure 1 shows the user interface of Jeliot 3 after the user has pressed the Fault button at the right time.

Conflictive animations can be used in assessment by asking students to press the Fault button whenever they detect a conflict. To avoid random trials, there is a limit, 3, to the number of times the Fault button can be pressed in a single run of the animation. If this limit is reached, the animation is restarted. In addition to this, the conflict may only be apparent to the students some time after it has happened. In Jeliot 3, students have a variable number of steps after which they can still report the occurrence of an error.

When the user presses the Fault button, she will be informed of the success of her trial. If unsuccessful, she will have to continue watching the animation looking for an error. If the students has pressed the Fault button at the correct time, a multiple choice question checks whether her reason for pressing it was the correct one. This helps the student to reflect better on what has happened and why; teachers could gather this valuable feedback to identify misunderstandings held by the students.

Finally, when a conflict animation has been spotted, Jeliot 3 rewinds itself and correctly animates the conflictive concept. On the contrary, if the animation reaches the end and the user has not spotted the error, she is given a hint of where the error is and asked to try again.

Awareness of the conflicts.

It is still not clear how we should make the students aware of the possible conflicts present in the animation. On one hand, students should be aware of the types of conflicts or their levels of abstraction that are possible in the current animation in order to be able to concentrate on right aspects of the program execution. Due to the interpretative nature of Jeliot 3, it can not detect and warn in advance of the errors that are going to happen.

4. DISCUSSION

In this paper we have presented an implementation of conflictive animations that addresses the responding level of the engagement taxonomy [6]. The responding activity corresponds to the act of students spotting in time when the animation has gone wrong. Higher levels in the taxonomy,as changing or presenting, are not suitable for automatic creation, but they should also be considered when working with conflictive animations.

It is worth noting, that according to our beliefs, spotting and identifying the error is not paramount. With conflictive animations students should now pay more attention to the animation and force themselves to understand the animation principles and the programming concepts presented with them.

Correct presentation and interaction with conflictive animations is an open topic, and only one possibility is implemented in Jeliot 3. Feedback from students is currently being analysed for different modes of interaction as well as effectiveness of the concept. We consider that automatic generation of conflictive animations opens the door to explore all the possibilities that conflictive animation can offer to improve students understanding of programming and their usage of visualization tools.

5. REFERENCES

[1] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen. The jeliot 2000 program animation system. *Comput. Educ.*, 40(1):1–15, 2003.

[2] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke. Matrixpro - a tool for demonstrating data structures and algorithms ex tempore. In *Proceedings of ICALT 2004*, pages 892–893, 2004.

[3] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing Program with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004*, pages 373–380, Gallipoli (Lecce), Italy, 2004.

[4] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller. Conflictive animations as engaging learning tools. In R. Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 203–206, Koli National Park, Finland, 2007. ACS.

[5] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. J. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally. Evaluating the educational impact of visualization. In *ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 124–136, New York, NY, USA, 2003. ACM.

[6] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152, New York, NY, USA, 2002. ACM Press.

Is Automatic Evaluation Useful for the Maturity Programming Exam?

Bronius Skupas
Institute of Mathematics and Informatics
Akademijos str. 4
LT-08663 Vilnius, Lithuania
+37068477349
bskupas@ktl.mii.lt

Valentina Dagiene
Vilnius University,
Naugarduko str. 24
LT-03225 Vilnius, Lithuania
+37069805448
dagiene@ktl.mii.lt

ABSTRACT

The optional maturity programming exam is considered as an outcome of the secondary curriculum on information technologies in Lithuania. The most important part of the exam is the evaluation of the students' programs. A special application was developed for automatic and manual evaluation of programs. It evaluates program correctness, programming constructs and style. The application proposes a score to evaluators who make the final decision. A comparison of evaluations shows potency in this area.

Keywords

programming examination, automatic evaluation, programming style.

1. PROGRAMMING EXAM AND A SCHEME OF ASSESSMENT

In Lithuania, informatics and information technologies (IT) were incorporated into the optional maturity examination block. A common discussion on the development of maturity examination in information technologies and informatics was presented in [1]. Considering that programming as a branch is fairly important to the state's economy, the decision was made to introduce a special examination in programming.

Since the content of the informatics curriculum puts an emphasis on information technologies, it was decided to develop two types of maturity examinations. The first one is intended to evaluate the students' skills of using information and communication technologies (ICT) at a school level. Another one is focused on programming skills and is intended to promote the professional studies (informatics, computer science, software engineering, etc.) in higher education.

The national examination in programming (since 2006) focuses on: knowledge and understanding – 30%, skills – 30%, and problem solving – 40%. The examination consists of test questions in IT (25%), test questions in programming (25%) and two practical programming tasks (50%).

The aim of the programming test is to examine the level of students' knowledge and understanding of the tools required in programming (elements of the programming language, data types and structures, control structures, basic algorithms).

The goal of practical tasks is to create programs for the given problems. Developing programs is one of the most important parts of that type of examination as well as one of the most difficult tasks for students.

In the sequel we will focus on programming tasks and their evaluation.

2. EVALUATION PROCESS

The problems selected for the programming part are oriented towards the selection of data structures and application of basic algorithms to work with the data structures created. Solutions of problems must be batch style programs: input data must be read from the text file and output should be made in another text file. Students have to write programs using the FreePascal programming language.

The criteria for the evaluation of programs have been found (Table 1). The proportions of points allocated for each particular task varied in rather a narrow range, depending on the particular difficulty and extent of the task.

Table 1. Programming evaluation criteria and weight in 2008

Evaluation criteria	%
Program testing result. If all the tests are positive, all points are marked, otherwise, the points are marked for reading from the file, correct parts of the program.	70
Subroutines, their parameter lists, correct calls to them, subroutines are performing actions required in the task.	18
Data types and variables. Suitability of data types for the task, their correctness and rationality. Correct use of variables.	4
Style of the programming. Accordance with the spelling rules, comments, structuring of the program's text layout, names of the constants, types and variables as well as their suitability for the task.	8

Testing of solutions of the exam was challenging. Prognosis was for 1500 students. We had some practice with Olympiads in

Informatics, but the amount of participants was lower there. The analysis of the exam has showed that evaluation will be quite different from that of Olympiads in Informatics. Usually quite good programmers participate in contests on programming and even there the scores are sometimes very low. It was clear, that automatic evaluation in examination should be very different from contests in evaluation of non-functioning programs.

The main difference between the concept of the Olympiad and that of maturity examination in programming is the idea of “fixing” small errors in programs. The problem with the “fixing” concept was that it is difficult to tell whether it is small error or big, and how many patches we can provide, etc. After patching we must retest the program with all data sets, and afterwards we think how many points a student lost with this error…

It has turned out that an interactive system is required for evaluation.

2.1 Program testing with data sets

Every solution must be compiled; afterwards it must be run with several data sets. In some tasks several different outputs can possibly be evaluated as correct. This gives an idea to use a different output correctness checker for each task.

As students are not professional programmers, it is common to have different simple errors in output format. Namely, forgotten spaces between numbers, all the output in one line, etc. could be examples of such style errors. A decision was made to split the correctness checker into two output evaluating programs: the output format checker and the output evaluator.

2.2 Automatic analysis of the source code and automatic evaluation of the programming style

Manual evaluation of the programming style is subjective. Several evaluators usually provide a different score. This caused an idea to create an automatic programming style evaluating application that can generate a reference score for evaluators. This application had to make analysis of used data structures and subroutines.

”Research on measurable programming style definitions was very active in the 1980’s” and is still in progress [3]. The main problems in this area are standards for the good programming style and choice of measure. Pascal programming style Rees [4] measures with simple statistics such as percentage of comment lines, number of gotos, and the average length of identifiers. Later P. W. Oman and C. R Cook [5] proposed taxonomy for programming style.

After the expert discussion it has been agreed that the main programming style concept will be based on the Charles Calvert (Borland employee) concept [2]. Later on, following criteria were selected for Pascal programming style evaluation: reasonable text indentation; spaces before keywords; same capitalisation for all variable, procedure names; definition and usage of procedures and functions; definition and usage of the record data type; definition and usage of the array data type; comment percentage.

After the source analysis we faced a long list of different error counters. The national examination schema was rather different from the set of these counters. How to convert a huge amount of data to only several numbers? The start was from the with Rees [4] range idea. According to it the best values for some factor are achieved in a specific range. If the values are outside of the range, they have a regression to low values. The obtained values, multiplied by weights and bias, were added to the sum of products. This technique was used in the experimental examination and showed a great potency in usage.

We have done several experiments with the data obtained by application and human evaluators. The correlation between manual and automatic evaluation is rather promising. It was between 0.61 and 0.72.

There were two types of exceptions. First type of exceptions was high scoring for very short (5-7 not empty lines) programs and second type was low score for non-functional solutions.

The program analyser was too sensitive to in case of syntax errors. Sometimes human evaluators fixed errors and marked them with high score. On the other hand, an automatic evaluator had problems with syntax and marked with very low score. We also reduced analyser sensitivity for syntax errors forced re-evaluation of the programming style after simple error fixing.

2.3 Discussion and problems

- Is it necessary to evaluate the programming style in programming examination?
- Is it possible to have a universal set of requirements for programs with a good programming style?
- What technologies can be used for the best human and automatic evaluator congruence?

3. ACKNOWLEDGEMENTS

Some activities in evaluating programs are part of the project "Integrating On-line Judge into effective e-learning" (UVa Online Judge) which has been funded from the European Commission under the grant number 135221-LLP-1-2007-1-ES-KA3-KA3MP.

4. REFERENCES

[1] Blonskis, J., Dagienė, V., 2008, Analysis of Students’ Developed Programs at the Maturity Exams in Information Technologies. Lecture Notes in Computer Science: Informatics Education – Supporting Computational Thinking: International Conference in Informatics in Secondary Schools – Evolution and Perspectives. 2008, Vol. 5090. p. 204-215.

[2] Calvert Ch. Object Pascal Style Guide, accessible at http://dn.codegear.com/article/10280.

[3] Ala-Mutka, K., Uimonen, T. , Järvinen, H.-M., 2004, Supporting Students in C++ Programming Courses with Automatic Program Style Assessment, Journal of Information Technology Education, Volume 3, 2004, accessible at http://jite.org/documents/Vol3/v3p245-262-135.pdf.

[4] Rees, M. J., 1982, Automatic assessment aids for Pascal programs. SIGPLAN Notices, 17 (10), 33-42.

[5] Oman, P. W., Cook, C. R., 1991, A programming style taxonomy, Journal of Systems and Software, v.15 n.3, p.287-301, July 1991.

"How a Contextualized Curriculum work in Practice"

Joseph M. Longino
Department of Information Technology, Faculty of Technology Management
Lappeenranta University of Technology
P.O. Box 20
Skinnarilankatu 34
FIN-53851, LAPPEENRANTA, Finland
joseph.longino@gmail.com

Mikko Vesisenaho
Research and Development Center for Information Technology in Education(TOTY),Faculty of Education,
University of Joensuu,
P.O. Box 111
FIN-80101 JOENSUU, Finland
mvaho@joyx.joensuu.fi

ABSTRACT

In developing countries higher learning institutions have been at the forefront of acquisition developing and use of technology. Most of institutions influence the dissemination and therefore accessibility of technology in their respective regions. Moreover the higher learning institutions have been at the forefront in producing the ICT skills in need within the respective communities and hence influence the knowledge and its application within the society. In developing countries such ICT skills being produced mostly focus on satisfying the various industrial ICT demands.
In this study we try to analyze the role of the higher learning institutions in the provision of ICT education and therefore assess the application of such educational curriculum to stimulate development and information access to information divisive communities in the developing countries.

Categories and Subject Descriptors

4 [**Educational technology, software, and tools**]

General Terms

Curriculum Evaluation

Keywords

Computer science education, Information and communications technology, Developing countries, Curriculum Evaluation, contextualization, Higher education, Tanzania.

1. INTRODUCTION

In Tanzania Information technology knowledge is vital to supplement the pace fast growing economic and development activities [4], which demands high and reliable level of expertise in computing field. On the other hand motivation for the research arose from challenges that many of the developing countries face today. One of these challenges is to utilize the education system as a means to provide advancement in learning and therefore to stimulate development within their local context [8].
In 2006, a purposeful research was carried out at Tumaini University to design a contextualized curriculum that can supplement for such needs hence facilitate development in Tanzanian context [6]. A contextualized curriculum took advantage of six principles namely curriculum contextualize, projects, practical, interdisciplinary orientation, international recognition and continuous research for the programs formative and development [5].
The program begun on September 2007, and implementation followed the CATI (Contextualize, Apply, Transfer, and Import) model with emphasis on students to identify societal expectations at the early stage in learning process, in which case the graduates will potentially cater for societal expertise needs on ICT [6].
Therefore the evaluative research aimed at understanding the process of implementation according to the six defined principles, evaluate contribution of the knowledge derived from the curriculum to societal development and finally suggesting a way forward to the future of the IT in a developing context.
This has a vital role in producing skills which sufficient quality to promote societal development and or improve the respective livelihoods in such communities [2].

2. RESEARCH APPROACH

On the basis of the research, as a single case study utilized a case analysis technique where by a phenomena was observed as a snapshot of its implementation time to reveal progress and commitment towards the set of objectives of curriculum under investigation [7]. In combination to this is an emergent exploratory, qualitative inquiry approach. The basis of selecting such a hybrid approach allows diversity in determining the limits of the study, which goes hand in hand with the fact that our behaviors are mostly determined by the influences surroundings [3].

Figure 1 shows the data collection and selection procedure. The field work was conducted on the actual setting where implementation had been taking place hence facilitating the collection of a variety of empirical material case study, personal experience, introspective, life story interview, observational, historical, interaction, and visual texts-that describe routine and problematic moments and meaning in individual lives.

Koli Calling '08 November 13-16, 2008, Koli, Finland
Copyright 2008 ACM 978-1-60558-385-3/08/11 $5.00.
.

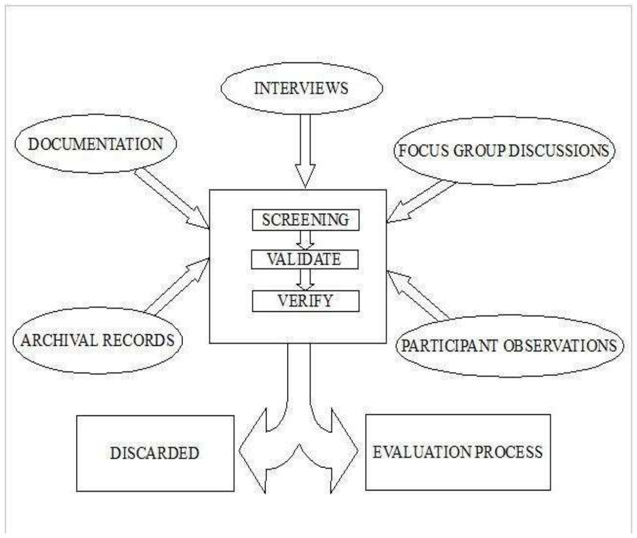


Figure 1: Data collection procedure.

3. RESULTS

Observation of the Constructs in Implementation of B Sc IT curriculum different participant groups of the B Sc IT curriculum observe the implementation process in different perspectives. From the chosen constructs observable by the study each group gave an inspiration to the study hence assisted to depict the reception of B Sc IT on its respective setting. Below is the table which shows the extent of satisfaction in implementation of B Sc IT curriculum by the different actors.

3.1 The six principles

Since implementation of B Sc IT curriculum is based on the six principles[1], it is therefore important to understand their effect and reflection by the participant groups in the implementation. The study revealed that to a larger extent the six principles had been satisfied through implementation. However, some parts which are considerably important were observed to be overlooked in the implementation process of which will be detailed in later sections.

3.2 Transition in learning

Students participated on the first phase of implementation join from other disciplines majority being from bachelors education (B. Ed)[1]. Students found it was interesting to join the B Sc IT program managed to do so provided they could meet the admission criteria and or pass the entry exam for B Sc IT program. The drive among majority of the students to join B Sc IT program is influenced by the trend of technology, and therefore mostly felt that it was more important to learn and later teach technology education than just conventional education and therefore be able to contribute to the ICT development within their respective communities through education.

3.3 Constructing foundational knowledge in the use of ICT

Smooth understanding of the ICT concepts and the respective E-learning environment namely moodle makes the inception of the B Sc IT program very successful. This is being facilitated by the availability of computing facilities which enables students to adapt and understand technology very fast hence improving their ICT awareness within a very short time from the commencement of the program.

However, its is still quite early to overlook for ambitions among students, for example if at all they are willing to go abroad to pursue further studies especially in advanced IT programs and therefore it is such a restriction to implementers to really know the precise level of skills that they can provide to such students. This is expected to have been learn t in the second year, after having identified students expectations and focus on their study in the B Sc IT program.

3.4 Inclusive information society

For a contextualized curriculum to accomplish its goals, it becomes necessary for implementation to focus on establishing and inclusive information society. The society aims at spreading the ICT knowledge to surrounding community and therefore promoting ICT awareness and creativity in solving basic problems using ICT. However ICT awareness also must clearly define the precise boundaries as to what kind of problems that ICT solutions can intervene as a useful tool in solving such problems.

3.5 Contextual framework in BSc IT Curriculum

Considering contextualize of B Sc IT curriculum, it works on different levels which are curriculum, topic, and pedagogical levels. Pedagogical approach has been different as compared to many other programs since it was contextualized specifically for Tumaini University. Through this, the university will be able to produce professionals who will be specialists to ICT problems within Tanzania and at the same time these compete, work and pursue their further study abroad as defined in the internationalization context.

3.6 Practical orientation

Implementation took a project based approach so that graduates will have sufficient hands on experience. Therefore, more efforts were made aiming at practical sessions which involves finding solutions to given project problems either in group work or at individual level. Through this approach it is therefore clear that students will be able to start thinking of their area of specialization in either academic or as employed professionals, earlier enough hence establish their career path.

3.7 Pedagogical thinking in the contextualized ICT education

Introduction of the E-learning environment has its vital impact in the learning process. Most students in the intake are used to the conventional lectures, and find it very important to have lecturers have lecturers present physically in finding the mutual understanding of knowledge being delivered. With E-learning incorporated within B Sc IT curricula, it therefore becomes important to change the pedagogical thinking among students. This enables them to understand why they are learning in the new and different ways, what do they benefit from learning in new ways, and also which new learning ways will fit to their learning context.

3.8 Emphasizing on internationalization of B Sc IT curricula

Implementation of B Sc IT curriculum emphasizes on internationalization, as defined on the six principles. Internationalization context defines the need importance of the B Sc IT curriculum to align with the international standards, which means that knowledge to be achieved by students should be recognized at the international level and in turn providing them with the right to pursue further studies and career both within national and international level. Students in the program receive their lectures and feedback from international teachers and or professors who have a remarkable experience on particular field in education and technology.

4. CONCLUSION AND RECOMMENDATIONS

Participant observation revealed the lacking of enough expertise by the tutors in the implementation which in way affects the process of knowledge transfer to students. In this way a more knowledgeable group staff with enough academic expertise will highly suffice for the tutoring and research on the same curriculum in order to improve and stabilize it.

Through the results of this research, our study realized a satisfactorily inspiration on the six principles as the main drivers of the curriculum. This had been achieved through efforts being made by the administration of the university which aims to promote the B Sc IT program to become a useful tool to bring about changes and development as well. However there had been some conflicts realized within the six principles which appear to be pulling the curriculum between the two extremes. The principles are internationalization and contextualize whereby the implementation is trying to contextualize ICT education and at the same time promoting the curriculum for international recognition which results to contraction if such constructs in implementation. Further more the study realized the importance of emphasizing, scheduling and promoting e-learning, practical session, and interactive tutoring in order to simplify the learning process and therefore successful in implementation. Finally the study recommends an iterative and longitudinal review research on the curriculum to promote its stabilization, efficiency and sustainability thus, more suitable and applicable in a developing context.

5. REFERENCES

[1] Tumaini University 2006. Tumaini University bachelor of science in information technology (BSc-IT) curriculum. *BSc IT Curriculum*, 1:6–7, 2006.

[2] Vesisenaho M. Sutinen E. Lund H.H. Contextual analysis of students learning during an introductory ict course in Tanzania. *Institute for Electronic and Eelectrical Engineers-Technology for Education in Developing Countries (IEEE-TEDC0́6),Retrieved on January 10th, 2008*, 4:9–13, 2006.

[3] Longino J.M. Evaluation of implementation of bsc it curriculum at tumaini university. Master's thesis, Faculty of Technology Management, Department of Information of Technology, 2008. http://https://oa.doria.fi/handle/10024/42444.

[4] Veen M. Mulder F. Lemmen K. What is lacking in curriculum schemes for computing/informatics? *Association for Computin and Macinery (ACM) SIGCSE Bulletin*, 36:186–190, 2004.

[5] Haapakorpi R. Lund H.H.(2007) Bangu N. Myller N. Ngumbuke F. Sutinen E. Vesisenaho M. Information technology degree curriculum in tanzanian context.in p. cunningham m. cunningham, m.(eds.). *Proceedings of Information society technologies Africa, IST-Africa 2007,May, Maputo,Mozambique, International Information Management Corporation (9 pages).*

[6] Vesisenaho M. *Developing University level Introductory ICT Education in Tanzania:A contextualized approach.* PhD thesis, Department of Computer Science and Statistics, 2007. ftp://cs.joensuu.fi/pub/Dissertations/vesisenaho.pdf.

[7] Yin R.K. *Case Study Research Design and Methods*, volume 3. SAGE publications, 2003.

[8] Mulder F. van Weert T. Ifip/unesco's informatics curriculum framework 2000 for higher education. *Association for Computing Machinery-SIGCSE, Retrieved on January 20th, 2008*, 33:75–83, 2001.

Author Index

Ahoniemi, Tuukka 80

Bangu, Nicholas 51
Berglund, Anders 76

Cheng, Ka Fai 96
Clear, Tony 41

Dagiene, Valentina 117
degraaff, erik 1
Denny, Paul 109

Haddow, John 96
Hamer, John 109
Hewner, Michael 72
Hill, Jonathan 41

Isomöttönen, Ville 25

Johnson, Colin 84

Kaila, Erkki 101
Kiesmüller, Ulrich 16
Knobelsdorf, Maria 62, 72
Kollanus, Sami 25
Korhonen, Ari 88
Koski, Marja-Ilona 32
Kurhila, Jaakko 32
Kurmas, Zachary 105

Lönnberg, Jan 76
Laakso, Mikko-Jussi 101
Lahtinen, Essi 92
Liu, Yong 41
Longino, Joseph 119
Luxton-Reilly, Andrew 109

Malmi, Lauri 76, 88
Moreno, Andrés 115
Myller, Niko 115

Ngumbuke, Fredrick 51

Pasanen, Tomi A. 32
Paterson, James 96
Pears, Arnold 41
Plimmer, Beryl 41
Poplawski, David 105
Purchase, Helen 109

Rajala, Teemu 101
Romeike, Ralf 113

Salakoski, Tapio 101
Skupas, Bronius 117
Sorva, Juha 5
Sutinen, Erkki 51

Taherkhani, Ahmad 88
Tedre, Matti 51

Vesisenaho, Mikko 119

Whalley, Jacqueline 41

Keyword Index

Adult Education 32
Adult Learning Theories 32
Algorithm recognition 88
Algorithms 16
Assessment 84
Associations 96
Automatic assessment 88,101
Automatic evaluation 117

Categorization 72
Computer Biographies 72
Computer Science Education 51
Computers and Society 72
Computing curricula 80
Computing education 113
Computing Education Research 41,62,72
Conflictive animation 115
Contextualization 51
Contextualized education 119
Contributing student pedagogy ... 109
Creativity 113
CS minor student 80
CS novices 62
CS1 5,115
Curriculum issues 119

Debugging 76
Didactics of Informatics 16
Different disciplines 80
Digital Ink 41

Education 25,105
Engineering Education Research 1
Ethnocomputing 51
Evaluation 32
Experimentation 62
Explanograms 41

Human Factors 62

IT Education 51

Java 96

Learning 1,92
Logic 105

MCQ 109

Novice programming 101

Patterns 96
Peer assessment 109
PeerWise 109
Phenomenography 5
Podcasts 84
Problem Solving Process 16
Program visualisation 92,101,115
Programming education 76,101
Programming examination 117
Programming style 117

Question test bank 109

References 5
Review 92
Robots 32

Secondary CS Education 16
Simulation 105
Software visualisation 76
Static program analysis 88
Stereotypes 72
Student development 1
Students understandings 5

Tablet PC 41
TDD 25
Teaching Experiment 32
Tool-Based Analysis 16
Typology 62

UML 96

Variables 5

UPPSALA
UNIVERSITET

February 2009
ISSN 1404-3203
http://www.it.uu.se/