

CONFERENCES IN RESEARCH AND PRACTICE IN  
INFORMATION TECHNOLOGY

VOLUME 88

KOLI CALLING 2007



AUSTRALIAN  
COMPUTER  
SOCIETY



# KOLI CALLING 2007

Proceedings of the  
Seventh Baltic Sea Conference on Computing Education  
Research,  
Koli National Park, Finland, 15-18 November 2007

Raymond Lister and Simon, Eds.

Volume 88 in the Conferences in Research and Practice in Information Technology Series.  
Published by the Australian Computer Society Inc.



Published in association with the ACM Digital Library.

**Koli Calling 2007.** Proceedings of the Seventh Baltic Sea Conference on Computing Education Research, Koli National Park, Finland, 15-18 November 2007

**Conferences in Research and Practice in Information Technology, Volume 88.**

Copyright © 2008, Australian Computer Society. Reproduction for academic, not-for-profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:

Raymond Lister  
Faculty of Information Technology  
University of Technology Sydney  
Australia  
E-mail: [raymond@it.uts.edu.au](mailto:raymond@it.uts.edu.au)

Simon

School of Design, Communication and Information Technology  
University of Newcastle  
Australia  
E-mail: [simon@newcastle.edu.au](mailto:simon@newcastle.edu.au)

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland  
John F. Roddick, Flinders University, South Australia  
Simeon Simoff, University of Technology, Sydney, NSW  
[crpit@infoeng.flinders.edu.au](mailto:crpit@infoeng.flinders.edu.au)

Publisher: Australian Computer Society Inc.  
PO Box Q534, QVB Post Office  
Sydney 1230  
New South Wales  
Australia.

Conferences in Research and Practice in Information Technology, Volume 88  
ISSN 1445-1336  
ISBN 978-1-920682-69-9

Printed April 2008 by Griffith University Uni Print, Nathan Campus, Kessels Rd. Nathan, 4111, QLD, Australia.  
Cover Design by Modern Planet Design, (08) 8340 1361.

The *Conferences in Research and Practice in Information Technology* series aims to disseminate the results of peer-reviewed research in all areas of Information Technology. Further details can be found at <http://crpit.com/>.

# Table of Contents

## Proceedings of the Seventh Baltic Sea Conference on Computing Education Research, Koli National Park, Finland, 15-18 November 2007

Preface .....	ix
Programme Committee .....	x

### Keynote Paper

Constructive alignment and the SOLO taxonomy: a comparative study of university competences in computer science vs. mathematics .....	3
<i>Claus Brabrand and Bettina Dahl</i>	

### Research Papers

What does “objects-first” mean? An international study of teachers’ perceptions of objects-first .....	21
<i>Jens Bennedsen and Carsten Schulte</i>	
Contextualising information and communications technology in developing countries .....	31
<i>Meurig Beynon, Anthony Harfield, and Mikko Vesisenaho</i>	
Incorporating programming strategies explicitly into curricula .....	41
<i>Michael de Raadt, Mark Toleman, and Richard Watson</i>	
An evaluation of electronic individual peer assessment in an introductory programming course .....	53
<i>Michael de Raadt, David Lai, and Richard Watson</i>	
Computer science in context — pathways to computer science .....	65
<i>Maria Knobelsdorf and Carsten Schulte</i>	
Students’ understandings of concurrent programming .....	77
<i>Jan Lönnberg and Anders Berglund</i>	
Applying creativity in CS high school education — criteria, teaching example and evaluation .....	87
<i>Ralf Romeike</i>	
Student transformative learning in software engineering and design: discontinuity (pre)serves meaning .....	97
<i>Leslie Schwartzman</i>	
ICT teaching and learning in a new educational paradigm: lecturers’ perceptions versus students’ experiences .....	109
<i>Judy Sheard and Angela Carbone</i>	
Koli Calling comes of age: an analysis .....	119
<i>Simon</i>	
Students’ understandings of storing objects .....	127
<i>Juha Sorva</i>	
Computer science students’ experiences of decision making in project groups .....	137
<i>Mattias Wiggberg</i>	

## System Paper

VILLE — a language-independent program visualization tool .....	151
<i>Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski</i>	

## Discussion Papers

Fighting the student dropout rate with an incremental programming assignment .....	163
<i>Tuukka Ahoniemi, Essi Lahtinen, and Teemu Erkkola</i>	
Improving mathematics and programming education — the IMPED initiative .....	167
<i>Ralph-Johan Back, Linda Mannila, Mia Peltomäki, and Tapio Salakoski</i>	
Debating the OO debate: where is the problem? .....	171
<i>Anders Berglund and Raymond Lister</i>	
A doctoral course in research methods in computing education research. How should we teach it? ...	175
<i>Anders Berglund, Päivi Kinnunen, and Lauri Malmi</i>	
Using topic map technology in the planning of courses from the CS knowledge domain .....	179
<i>Evgeny A Eremin</i>	
Exploration module for understanding the functionality of the internet in secondary education .....	183
<i>Stefan Freischlad</i>	
Should we assess our students' attitudes? .....	187
<i>Ursula Fuller and Bob Keim</i>	
Puck — a visual programming system for schools .....	191
<i>Lutz Kohl</i>	
Effectiveness of integrating program visualizations to a programming course .....	195
<i>Essi Lahtinen, Tuukka Ahoniemi, and Anniina Salo</i>	
CLIP, a Command Line InterPreter for a subset of C++ .....	199
<i>Harri Luoma, Essi Lahtinen, and Hannu-Matti Järvinen</i>	
Conflictive animations as engaging learning tools .....	203
<i>Andrés Moreno, Erkki Sutinen, Roman Bednarik, and Niko Myller</i>	
What's the problem? Teachers' experience of student learning successes and failures .....	207
<i>Arnold Pears, Anders Berglund, Anna Eckerdal, Philip East, Päivi Kinnunen, Lauri Malmi, Robert McCartney, Jan-Erik Moström, Laurie Murphy, Mark Bartley Ratcliffe, Carsten Schulte, Beth Simon, Ioanna Stamouli, and Lynda Thomas</i>	
Computer science students can help to solve problems of multiplayer mobile games .....	213
<i>Carolina Islas Sedano, Ekaterina Kuts, and Erkki Sutinen</i>	
Applications of variation theory in computing education .....	217
<i>Jarkko Suhonen, Errol Thompson, Janet Davies, and Kinshuk</i>	
How does internationalisation affect learning and teaching of computer science: a study at Tongji University in China .....	221
<i>Doris Dongsheng Yang and Anders Berglund</i>	

## Demo/Poster Papers

Why should we bore students when teaching CS? .....	227
<i>Tuukka Ahoniemi, Essi Lahtinen, and Keeko Valaskala</i>	
Nalkki-project — tool for plagiarism detection using the web .....	229
<i>Petri Sirkkala and Sami Puonti</i>	
<b>Author Index</b> .....	231



## Preface

The Seventh Baltic Sea Conference on Computing Education Research, Koli Calling 2007, was held in Finland's beautiful Koli National Park on 15-18 November 2007.

Contributions to Koli Calling can take one of four forms. Research papers present unpublished original research. System papers describe tools for learning, instruction, or assessment in computing education, motivated by the didactic needs of computing. Discussion papers are shorter papers used to present novel ideas, proposals, prototypes, or work in progress. Posters are very short presentations describing novel approaches or work in progress.

All papers and posters were double-blind peer reviewed by members of the international programme committee and additional reviewers. There were 30 'long' papers submitted (i.e. research or system papers). The acceptance rate for long papers was 43% (12 research papers and one system paper). The remaining long papers, and those papers submitted specifically as discussion papers, comprised a total of 28 papers, of which 15 (54%) were accepted as short papers. The authors of all these accepted papers are from 12 different countries.

The growth of Koli Calling is to a large degree due to the open and friendly atmosphere that encourages participants to return in subsequent years. There were 46 participants from ten countries across three continents, and most of these had attended the conference in previous years. A feature of Koli Calling is that all sessions are plenary, allowing discussion of all the work presented to continue through the breaks and into the evenings. The conference recognises the value of this discussion by encouraging authors to revise their papers after the conference for the proceedings. The authors worked closely with the proceedings editor to further improve the quality of the published proceedings. It is indicative of the growing internationalisation of Koli Calling that this year both the programme chair and the proceedings editor are based in Australia.

Practical arrangements for the conference were made by the Department of Computer Science, University of Joensuu, Finland. Particular thanks go to the local organiser, Ilkka Jormanainen.

When citing papers at this conference, please use the following format: <Author(s)> (2008). <Title>. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Koli National Park, Finland, November 2007. Conferences in Research and Practice in Information Technology, vol 88.

We thank everyone who contributed to the special atmosphere of Koli Calling 2007. In particular, we thank the programme committee for their dedication to providing constructive criticism of the submissions, and the organising committee who all worked very hard to ensure the conference was a success. However, most important is the contributions of the authors and the participants, without whom we would not have had a conference. We hope to see you all again at Koli Calling 2008.

Raymond Lister, Conference Chair  
Simon, Proceedings Editor  
March, 2008

# Programme Committee

## Programme Chair

Raymond Lister (University of Technology Sydney, Australia)

## Programme Committee

Anders Berglund (Uppsala University, Sweden)  
Angela Carbone (Monash University, Australia)  
Michael E Caspersen (University of Aarhus, Denmark)  
Valentina Dagiene (Vilnius University, Lithuania)  
Mike Joy (University of Warwick, UK)  
Ari Korhonen (Helsinki University of Technology, Finland)  
Lauri Malmi (Helsinki University of Technology, Finland)  
Robert McCartney (University of Connecticut, USA)  
Arnold Pears (Uppsala University, Sweden)  
Guido Röβling (Darmstadt University of Technology, Germany)  
Tapio Salakoski (University of Turku, Finland)  
Carsten Schulte (Free University of Berlin, Germany)  
Judy Sheard (Monash University, Australia)  
Jarkko Suhonen (University of Joensuu, Finland)  
Erkki Sutinen (University of Joensuu, Finland)

## Additional Reviewers

Michael de Raadt (University of Southern Queensland, Australia)  
Ilona Box (University of Technology Sydney, Australia)  
Anna Eckerdal (Uppsala University, Sweden)  
Logan Muller (Unitec, New Zealand)  
Matt Bower (Macquarie University, Australia)

## Proceedings Editor

Simon (University of Newcastle, Australia)

# KEYNOTE PAPER



# Constructive Alignment and the SOLO Taxonomy: A Comparative Study of University Competences in Computer Science vs. Mathematics

**Claus Brabrand**

IT University of Copenhagen, Denmark

brabrand@itu.dk

**Bettina Dahl<sup>1</sup>**

University of Aarhus, Denmark

bdahls@si.au.dk

## Abstract

In 2007, all Danish university syllabi were reformulated to explicitly state course objectives to comply with a new Danish national grading scale, which stipulated that grades were to be given based on how well students met explicit course objectives. This paper analyzes 550 syllabi from the science faculties at University of Aarhus, Denmark (AU) and the University of Southern Denmark (SDU) that had been rewritten to explicitly incorporate course objectives, interpreted as intended learning outcomes (ILOs), using the principles of Constructive Alignment and the SOLO Taxonomy. In this paper we explain and discuss these principles, give examples of how the new syllabi were constructed, and describe the process by which they were formed. We also explain and discuss the results of a comparative study comparing the competences of Computer Science with those of Mathematics (and classical Natural Sciences for a point of reference). In this study, we focus on what specific competences the respective departments primarily use.

**Keywords:** Constructive Alignment; SOLO Taxonomy; Competences; Intended Learning Outcomes (ILOs); Computer Science; Mathematics; Natural Science.

## 1 Introduction

This paper is an analysis of a data set consisting of 632 course syllabi from the science faculties at University of Aarhus, Denmark (AU) and the University of Southern Denmark (SDU). Both faculties have been through a process of formulating intended learning outcomes (ILOs) to existing course syllabi. The reason for this process was the adoption of a new nationwide Danish grading scale, which stipulates that grades are to be given based on how well students meet explicit course objectives. ILOs were thus formulated for all courses in the two faculties for one academic year. However, this paper will comparatively investigate only those of Computer Science, Mathematics, and classical Natural Sciences (here restricted to Physics, Chemistry, Biology,

and Molecular Biology), using the SOLO Taxonomy as a tool for analyzing ILO competences. In total, this gives us a data set of 550 courses. All ILOs have been formulated according to the principles of Constructive Alignment and using the SOLO Taxonomy. The academic staff at AU received a course on these principles by a group of five people, appointed by the dean, of which both authors were members and which was chaired by Brabrand. SDU had a very similar process for which Brabrand was a consultant. The paper is both a summary of the keynote talk given by Brabrand at Koli Calling 2007 and a further study of the data. The first part of the paper introduces the Theory of Constructive Alignment and the SOLO Taxonomy. The second part explains our comparative study of competences.

## 2 Constructive Alignment and the SOLO Taxonomy

### 2.1 The Theory of Constructive Alignment

The Theory of Constructive Alignment (Biggs 2003) is a theory of teaching and learning developed by John Biggs. It is a *systemic theory* in the sense that the entire teaching context is perceived as a 'system' for which we need to understand the individual parts and how they interact in order to understand and make predictions about the entire system. The theory is also based on the principles of *constructivism*, that knowledge is personal and that meaning is actively constructed by the learners themselves through active engagement with the subject matter. This perspective is in sharp contrast to the (once commonly held) idea that knowledge is 'transmitted' *from an active teacher to a passive student*. Finally, it is a *constructive theory* in the sense that it embodies constructive advice for what teachers ought to do in order to make sure their students learn what they intend. The 'solution' to this important challenge of teaching is for the teacher to *constructively align* courses 'ahead of time' (see Figure 1).

Before we can understand and appreciate why this is an

Definition: A course is said to be [constructively] aligned when:

- intended learning outcomes (ILOs) are explicitly formulated as operational competences;
- the ILO competences are explicitly communicated to the students (early in the course);
- the exams measure precisely the ILO competences; *and*
- the teaching/learning activities (TLAs) match the ILO competences.

Figure 1: Definition of 'constructively aligned course'

<sup>1</sup> Full name: Bettina Dahl Søndergaard (Soendergaard).

Goal:	<p><i>The goal of the course is to:</i></p> <ul style="list-style-type: none"> <li>- <b>introduce</b> students to general design techniques for the construction of effective algorithmic solutions to combinatoric problems; and</li> <li>- <b>familiarize</b> the student with effective solutions to important graph and string problems.</li> </ul>
Content:	<ul style="list-style-type: none"> <li>- algorithmic paradigms: <i>'divide and conquer', 'dynamic programming', 'greedy algorithms'</i>;</li> <li>- graph algorithms: <i>'traversal strategies', 'connectivity', 'topological sorting', 'spanning trees', 'shortest path', 'transitive closure'</i>; and</li> <li>- text processing: <i>'pattern recognition'</i>.</li> </ul>

Figure 2: Content description for 'Algorithms & Datastructures II' (Computer Science, AU)

interesting thing to do, we first (in the spirit of systemic theories) need to address three dependent factors: *teacher's intention*, *student's activity*, and *exam's assessment*. Please note that throughout the paper we will use the word 'exam' in a broad sense for any assessment activity during and/or after the course that counts towards a student's final grade; i.e. not just one 'all-or-nothing test' at the end of a course.

#### *Teacher's intention*

The ultimate goal of any teaching situation is that the students learn whatever it is they are supposed to learn. However, 'learn' is an inherently vague concept and may ambiguously refer to both 'learn (about)' and 'learn (to do)'. To 'learn *about* programming' is clearly very different from to 'learn to *do* programming' (i.e., 'to learn to program'). Traditionally, many teachers have created course descriptions formulated and communicated to students in terms of 'content' to be learned (about). Figure 2 is an example of such a course description for the undergraduate Computer Science course 'Algorithms & Datastructures II' from the University of Aarhus before the faculty-wide rewriting of all syllabi to include ILOs using the principles of constructive alignment and the SOLO taxonomy.

Teachers and examiners, being part of the same research-based teaching traditions, will most likely know immediately from the description in Figure 2 what it is the students are expected to be able to do when they are assessing. They will interpret (and most likely agree) that what is really meant by '*algorithmic paradigms*' is that a student is expected to (be able to), for example, '**construct** algorithms' and '**analyze** algorithms' using standard algorithmic paradigms. However, this is tacit

knowledge, usually not known by the students. Students, not part of the same research-based educational tradition, might interpret '*algorithmic paradigms*' in an altogether different sense; for instance, that they should be able to **name** standard algorithmic paradigms and **recite** running-times of textbook algorithms from each of the algorithmic paradigms, on command. Clearly, to 'construct and analyze' is qualitatively different from, and somehow at an entirely different level from, to 'name and recite'. The nature of this difference is precisely what is accounted for by the SOLO Taxonomy, which we will explain in some detail below.

Furthermore, without quantum leap advances in brain scanning technology, we simply cannot *measure* 'understanding'; how much 'knowledge' a student has, how well a student has been 'introduced' to, or how 'familiar' a student is with, a given concept such as 'algorithmic paradigms'. These are internal cognitive structures inside the brain, biochemical high-level structuring of which we know very little and on which we can currently only speculate. What we can do, however, is to have a student *do* something, and then measure the *product* and/or *process*. The keyword here is 'operationality' which is the other aspect captured by the SOLO Taxonomy in that the competences it taxonomizes are operational and measurable. 'Understanding', 'knowledge', and 'familiarity' are inherently non-operational and non-measurable goals. Before we turn to how the above course description would have been authored if adhering to the principles of constructive alignment, we need to consider the students; in particular, a student's activity (before, during, and after teaching).

#### *Student's activity*

The Susan and Robert dichotomy, conceived by John Biggs (Biggs 2003), fits students to models (also known as personas) according to their motivation for studying at university. The personas should be thought of as prototypical student strategies rather than actual individuals. Figure 3 depicts Susan and Robert as *personified* in the 19-minute award-winning short film about constructive alignment, entitled 'Teaching Teaching & Understanding Understanding' (Brabrand & Andersen 2006).

Susan (Figure 3a) is intrinsically motivated and at university to learn: "Susan likes to get to the bottom of things; to reach understanding. She often reflects on

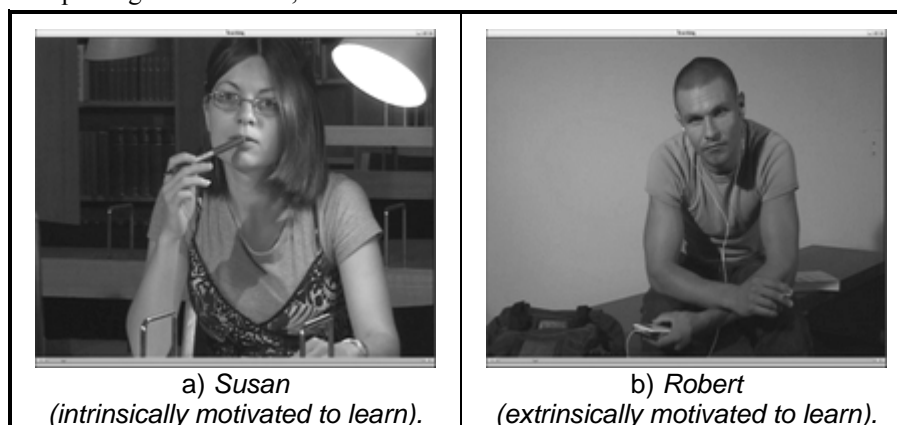


Figure 3: The Susan and Robert dichotomy introduced by John Biggs

What is the asymptotic complexity of 'topological sorting' on a directed graph $G = (V, E)$ ?		
<input type="checkbox"/> a)	$\Theta(\log( V  +  E ))$	i.e., "logarithmic time" in the size of the input
<input type="checkbox"/> b)	$\Theta( V  +  E )$	i.e., "linear time" in the size of the input
<input type="checkbox"/> c)	$\Theta(( V  +  E ) * \log( V  +  E ))$	i.e., "n-log-n time" in the size of the input
<input type="checkbox"/> d)	$\Theta(( V  +  E )^2)$	i.e., "quadratic time" in the size of the input

Figure 4: Hypothetical algorithmic MCQ leading to 'content memorization'

possibilities, implications, applications, and consequences of what she is learning. Susan is characterized by a preference for deep learning. She spontaneously uses higher cognitive processes. Faced with a curriculum, she basically teaches herself. In fact, we almost cannot prevent her from learning" (Brabrand & Andersen 2006).

Robert (Figure 3b) is extrinsically motivated and not at university to learn: "In fact, Robert doesn't really care about the learning in itself. His goal at university is different; his goal is [...] to pass exams, get a degree, and get a (decent) job. Robert is characterized by a preference for surface learning. He will only use higher cognitive processes if he really really really has to. He will cut any corner in achieving his goal with minimum effort" (Brabrand & Andersen 2006). Robert will stick with lower-level activities such as identification and memorization as long as they suffice.

As teachers, we do not need to worry about Susan; she will do fine. But what about Robert; what can we do to have him start acting more like Susan? Before we show how constructive alignment can be used to do just that, we need to look at the exam's assessment.

#### Exam's assessment

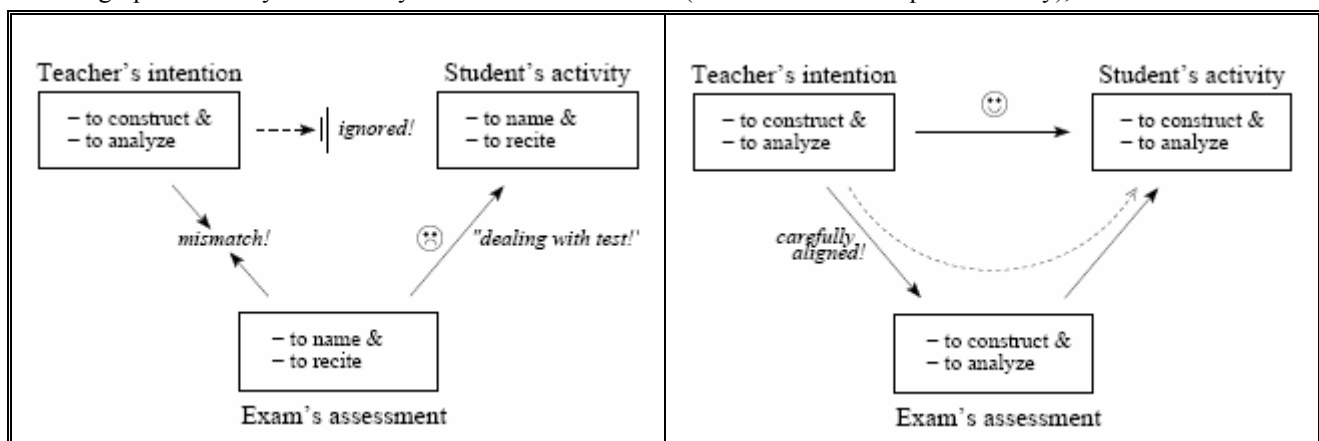
For many teachers and students, exams are a 'necessary evil'. However, the exam is perhaps the single most powerful pedagogical motivational tool available to teachers (and students) in that the exam has a constitutional effect on learning; the so-called 'backwash effect'. The exam has ramifications on how the students approach learning; in particular, on how willing they are to engage in learning activities. This includes any learning activity, whether in formally situated learning contexts planned by a teacher or autonomous informal learning spontaneously initiated by the student. "To the

teacher, assessment is at the end of the teaching-learning sequence of events, but to the student it is at the beginning" (Biggs 2003, p. 141). As an illustration of this, let us consider the (hypothetical) algorithmic multiple-choice question shown in Figure 4, which seems perfectly innocent.

When featured on an exam, however, the question will in essence reward students for using 'content memorization' strategies. The Roberts will do what is appropriately known as 'dealing with the test'; i.e., they will disregard the teacher's intentions of deep learning, and instead direct their learning towards strategic memorization of running times of textbook algorithms. They will thus stick with their low-level surface learning techniques (e.g., identification and memorization). In essence, we have what is known as an *unaligned* course.

Figure 5 illustrates the essential difference between an *unaligned* and an *aligned* course. (The figure abstracts away the issues pertaining to the teaching/learning activities, which we will address later.) In an *unaligned* course, we have a mismatch between the teacher's intention and the exam's assessment. The teacher intends for students to learn to 'construct and analyze', but the exam measures the competences to 'name and recite'. As outlined above, Robert will focus only on the skills required for the test, and disregard the teacher's intentions. In such courses the Roberts will seem disinclined to participate and engage in higher-level learning activities.

The solution to this problem proposed by John Biggs with his Theory of Constructive Alignment is to *constructively align* courses. The teacher is to formulate ILOs as operational competences from the SOLO Taxonomy (which we will explain shortly), communicate these



a) An unaligned course

b) An aligned course

Figure 5: An unaligned vs. aligned course (reproduced from Brabrand, 2007)

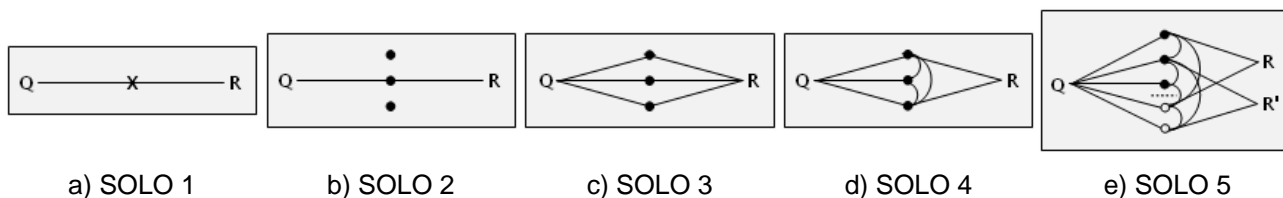


Figure 6: Visualization of the SOLO-levels 1-5. (Based on Biggs and Collis (1982, pp. 24-25).)

explicitly to the students early in the course, and meticulously design the exam such that it measures precisely those ILOs (and convince the students that this is indeed the case). The result is a ‘commuting diagram’ (Figure 5b) and teaching system where Robert’s desire to pass the course invariably leads him through learning the ILO chosen by the teacher.

#### Teaching/Learning Activities (TLAs)

Now Robert has the necessary *incentive* to learn, but we also need to provide appropriate *support* for him to learn effectively. This is where the teaching/learning activities (TLAs) fit in constructive alignment. The challenge is to choose TLAs that are likely to bring about acquisition of the ILO competences. Framing teaching activities as ‘training towards the exam’ will help engage and motivate Robert to participate actively.

## 2.2 The SOLO Taxonomy

The SOLO Taxonomy (short for *Structure of the Observed Learning Outcome*) originates from the study of student learning *outcomes* in university teaching carried out by John Biggs and Kevin F. Collis in the early 1980s. The taxonomy distinguishes five different levels according to the cognitive processes required by students in order to obtain them. “SOLO describes a hierarchy where each partial construction [level] becomes a foundation on which further learning is built” (Biggs 2003, p. 41). As described above, the taxonomy can be appropriately used to define ILOs in implementing Constructive Alignment. It is constructed particularly for research-based university teaching and converges on production of new knowledge (at its fifth and highest level) which is also the purpose and product of research itself. The five levels are visualized in Figure 6 and explained in the following, in increasing order of structural complexity (Biggs & Collis 1982, pp. 17-31; Biggs 2003, pp. 34-53). The figure depicts which elements are involved when a student at a given SOLO-level, given a *problem/question/cue* “Q”, produces an *outcome/response* “R” using the kinds of *data* either provided or not. The symbol “x” stands for irrelevant data; “●” stands for known related data that has been given to the student; and “○” stands for hypothetical related data that has not been given to the student.

#### SOLO 1: The Pre-Structural Level

Figure 6a depicts the outcome formation at SOLO level 1; a student is given a question or cue “Q” and uses irrelevant data “x” in producing a response, “R”. At this level, a student does not have any kind of understanding but uses irrelevant information and/or misses the point altogether. Scattered pieces of information (i.e., “x”) may have been acquired, but at this level they will be

unorganized, unstructured, and essentially void of actual content or relation to a relevant issue or problem, “Q”. Of course, students such as Robert may attempt to camouflage their lack of knowledge, by using ‘tautological responses’ i.e., reusing and rearranging *cues* of a question to produce an answer with essentially the same information that was embedded in the question, giving the illusion of understanding.

#### SOLO 2: The Uni-Structural Level

Level two is depicted in Figure 6b; a student is now capable of dealing with *one* relevant known aspect or issue “●” and use it in producing a valid but simple response “R”. Thus, from level one to two, we see improvements as the student becomes able to discern relevant issues and deal with *one* of these in relation to a problem “Q”. At this level, a student is capable of making obvious relevant connections and can, for instance, use correct terminology, remember things, recite, carry out simple instructions, identify, name, count, paraphrase on a sentence level, etc.

#### SOLO 3: The Multi-Structural Level

From level two to three we see *quantitative* improvements as the student becomes able to deal with a multiplicity of relevant known issues “●”. As illustrated in Figure 6c, a student is now capable of dealing with several aspects, but these are considered independently and not in connection to one another; e.g., how they may interrelate to form a whole. Metaphorically speaking, the student sees the many trees, but not the wood. He is able to enumerate, describe, classify, combine, apply methods, structure, execute procedures, etc.

#### SOLO 4: The Relational Level

At level four (Figure 6d), we begin to see *qualitative* improvements as the details integrate to form a structure. A student may now perceive relations between several aspects and how they might fit together to form a whole and structured response “R”. The student now sees how the many trees together form a wood. A student may thus have the competence to compare, relate, analyze, apply theory, explain in terms of cause and effect, etc.

#### SOLO 5: The Extended Abstract Level

From level four to five, we see further *qualitative* improvements as the structure is generalized and the student becomes capable of dealing with hypothetical information that was not given, “○” (Figure 6e). At this fifth and highest level, a student may now perceive the knowledge structure from many different perspectives and produce multiple responses (“R” and “R’”), depending on the perspective and hypothetical information included. Here, a student may have the competence to generalize, hypothesize, criticize, theorize, or transfer a theory to a new domain, etc.

- Quantitative -		- Qualitative -	
SOLO 2 <i>uni-structural:</i>	SOLO 3 <i>multi-structural:</i>	SOLO 4 <i>relational:</i>	SOLO 5 <i>extended abstract:</i>
<ul style="list-style-type: none"> <li>- paraphrase</li> <li>- define</li> <li>- identify</li> <li>- count</li> <li>- name</li> <li>- recite</li> <li>- follow (simple) instructions</li> <li>- ...</li> </ul>	<ul style="list-style-type: none"> <li>- combine</li> <li>- classify</li> <li>- structure</li> <li>- describe</li> <li>- enumerate</li> <li>- list</li> <li>- do algorithm</li> <li>- apply method</li> <li>- ...</li> </ul>	<ul style="list-style-type: none"> <li>- analyze</li> <li>- compare</li> <li>- contrast</li> <li>- integrate</li> <li>- relate</li> <li>- explain causes</li> <li>- apply theory (to its domain)</li> <li>- ...</li> </ul>	<ul style="list-style-type: none"> <li>- theorize</li> <li>- generalize</li> <li>- hypothesize</li> <li>- predict</li> <li>- judge</li> <li>- reflect</li> <li>- transfer theory (to new domain)</li> <li>- ...</li> </ul>
a) SOLO 2 competences	b) SOLO 3 competences	c) SOLO 4 competences	d) SOLO 5 competences

Figure 7: Prototypical verbs according to the SOLO Taxonomy; based on Biggs (2003, p. 48)

The terms ‘*surface understanding*’ and ‘*deep understanding*’ (also known as ‘*surface learning*’ and ‘*deep learning*’) are often used and, in fact, easy to define in conjunction with the SOLO Taxonomy. Surface learning implies that the student is confined to action at the lower SOLO levels (2-3); whereas deep learning implies that the student can act at any SOLO level (2-5), including the higher levels (4-5). Hence a student producing a high-level response (at SOLO 4-5) is thus often deemed to have a deep understanding of the matter. On the other hand, a student producing a lower level response (at SOLO 2-3) does not necessarily have a surface understanding. Finally, levels 2 and 3 are sometimes referred to as *quantitative* levels, and levels 4 and 5 as *qualitative*. Figure 7 lists prototypical competences from the SOLO Taxonomy, many of which were mentioned above.

### 2.3 Alignment implementation process

Figure 8 below shows the alignment implementation process as it was recommended to the teachers at the faculty courses at AU and SDU. It is explained in the following.

#### 1. Determine overall goals

The first step in designing an aligned course is to consider what are the overall things that the students are to get out of attending the course. Here, it is important to think in terms of competences *in addition to* content (the latter being what teachers are used to); i.e., what is it the students are supposed to learn to be able to *do* with the

content once the course is over?

#### 2. Operationalize goals as intended learning outcomes (ILOs)

The next step is to operationalize these goals and express them as ILOs in terms of the SOLO Taxonomy. This step can initially be a challenge for teachers.

#### 3. Choose forms of assessment (relative to ILOs)

Once the ILOs are chosen, the teacher needs to provide adequate *incentive* in order for students to learn the relevant competences (ILOs). Here, the teacher chooses / designs one or more forms of exams to cover all ILOs so that the competences are measured as precisely as possible. Certain combinations of ILOs and forms of exam are obvious mismatches (e.g., have an MCQ test to assess the competence ‘to explain’), but in most cases the teacher needs to carefully judge what exam form best fits the ILOs. This can also sometimes be a challenge due to practical issues and external constraints at the university, e.g., space, time, and/or economic issues.

#### 4. Choose forms of teaching (relative to ILOs)

With the ILOs in place, the teacher needs to provide adequate *support* for students to learn the relevant competences (ILOs). Here, the teacher may choose several teaching activities to cover all ILOs. Again, certain combinations of ILOs and teaching activities obviously do not go together; e.g., ‘lecture on (about) programming’ vs. ILOs stating ‘learning to program’ (as in ‘learning to do programming’). Again, this calls for careful judgements on behalf of the teacher. Steps 3 and 4 could be carried out in either order, or in parallel, but

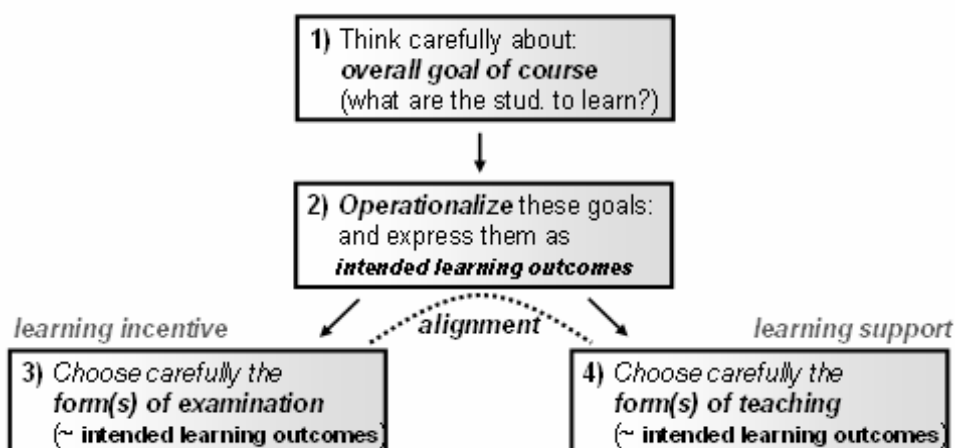


Figure 8: Alignment implementation process

At the end of the course, the student is expected to be able to...:
- <b>construct</b> (SOLO 4) and <b>analyze</b> (SOLO 4) algorithms using standard algorithm paradigms;
- <b>identify</b> (SOLO 2) and <b>formulate</b> (SOLO 3) algorithmic problems as graph and string problems;
- <b>identify</b> (SOLO 2) and <b>compare</b> (SOLO 4) graph and string algorithms for solving algorithmic problems;
- <b>construct</b> (SOLO 4) algorithms for simple graph and string problems.

Figure 9: Example ILOs from ‘Algorithms & Datastructures II’ (undergrad Computer Science)

aligning the teaching/learning activities towards the exam, ‘training towards the exam’, is likely to make the courses seem relevant to all students, including Robert; hence, it is often instructive to settle the form of exam prior to designing relevant TLAs. Alignment is then a product of how well steps 2, 3, and 4 correspond to one another. For more information on how to carry out this process and implement alignment for a specific science course, we refer to Brabrand (2007).

### 3 A Comparative Study of Competences in Computer Science vs. Mathematics

We now turn to a presentation and discussion of the specific competences used in Computer Science and Mathematics. Our data material for comparing these two subjects in depth according to their competences consists of 550 course syllabi. For an analysis of i) competence progression, ii) overall differences between science subjects, and iii) differences between similar departments at different universities, using the full 632 course data set, we refer to Brabrand & Dahl (2008).

Each of the 550 courses has a number of goals, each with a number of ILO competences. Figure 9 illustrates the competence description for the undergraduate Computer Science course ‘Algorithms & Datastructures II’ at AU.

The syllabi were created using the alignment implementation process (Figure 8). We have chosen to focus on the *formulated* ILOs because these have a strong impact on the grading since grades are to be given based on how well students meet the ILOs. The formulated outcomes are not necessarily always the same as the *formal*, *realized* (*operationalized*), or *learned* outcomes (Bauersfeld 1979; Goodlad 1986) but owing to the constitutional effect of examination on learning (‘teaching to the test’), the ILOs have an impact on the learning and in the event of students complaining about

Competence	Level	Frequency
- identify	SOLO 2	(2x)
- formulate	SOLO 3	(1x)
- construct	SOLO 4	(2x)
- analyze	SOLO 4	(1x)
- compare	SOLO 4	(1x)

Figure 11: Frequency counts for ‘Algorithms & Datastructures II’

grades, it is legally the formulated outcomes that matter. Thus, teachers are forced to take the formulated outcomes very seriously.

There are three ways in which we analyzed the data:

#### 1. SOLO average

We calculated a ‘SOLO average’ using a ‘double-weight averaging scheme’ in which each of the ILOs weigh the same and each of the verb competences within an ILO also weigh the same:

$$\left[ \frac{(4+4)}{2} + \frac{(2+3)}{2} + \frac{(2+4)}{2} + 4.0 \right] / 4 = 3.38$$

One might question whether in practice these ILOs do weigh the same. However, under the guidance of the group appointed by the dean that included the two authors, each department has formulated ILOs from a number of ‘standard good examples’ provided in advance. Hence if there is a variation, it is consistent throughout the syllabi. The idea of a SOLO average also rests on the assumptions that the ‘competence distance’ from, say, SOLO 2 to 3, is the same as between SOLO 3 and 4 etc. Such an approach of quantifying qualitative data is in fact often seen in educational research, for instance in the use of Likert scales that quantify degrees of agreement or disagreement using numbers, usually 1-5 (Oppenheim 1992; Robson 2002). Oliver et al. (2004) have carried out an analysis similar to ours using the six levels of the Bloom Taxonomy, but for only a handful of courses.

2. *SOLO distribution* The SOLO average will be complemented with comparisons of the relative distributions of SOLO levels for both individual and collective courses. One such example is seen in Figure 10, which shows the SOLO distribution of the above-mentioned course.

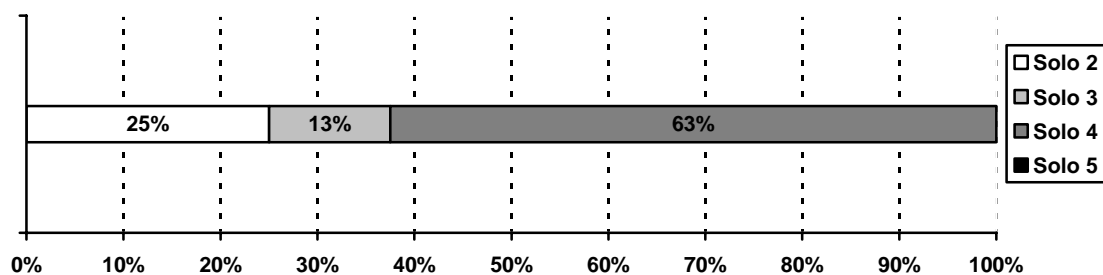


Figure 10: Relative distribution of SOLO levels for ‘Algorithms & Datastructures II’

- Quantitative -		- Qualitative -	
SOLO 2 <i>uni-structural:</i>	SOLO 3 <i>multi-structural:</i>	SOLO 4 <i>relational:</i>	SOLO 5 <i>extended abstract:</i>
- identify (168x)	- describe (677x)	- explain (382x)	- discuss (212x)
- calculate (80x)	- account for (593x)	- analyze (281x)	- assess (125x)
- reproduce (64x)	- apply method (485x)	- compare (103x)	- evaluate (58x)
- arrange (56x)	- execute proc. (154x)	- argue (75x)	- interpret (51x)
- decide (32x)	- formulate (85x)	- relate (70x)	- reflect (39x)
- define (25x)	- use method (75x)	- implement (55x)	- perspectivate (37x)
- recognize (20x)	- solve (68x)	- plan (44x)	- predict (28x)
	- conduct (61x)	- summarize (35x)	
	- prove (57x)	- construct (31x)	
	- classify (36x)	- design (21x)	
	- complete (34x)		
	- combine (25x)		

Figure 12: Examples of verbs at SOLO levels  
(verbs occurring at least 20 times in course descriptions at AU & SDU)

### 3. SOLO frequencies

A third way is to see the frequencies, either in raw numbers or in percentages of how often various competences within each SOLO level occur. Figure 11 shows the list for the same example course.

Looking at the whole dataset, the list becomes as shown in Figure 12, which lists the verbs / SOLO competences occurring at least 20 times in the syllabi of all 632 courses. We chose to cut at 20 to give an overview of the main competences used more than just a few times.

The validity of our analysis depends first on SOLO being an appropriate description of competences and second on our SOLO classification being appropriate. In relation to the former, we built our work on the SOLO model, which is the result of extensive research done by Biggs and Collis (1982, 2003); with regard to the latter, through an iterative process of three stages we consulted several other educational researchers (including Biggs) to get feedback on the classification, which resulted in Figure 12. The three approaches to data analysis complement each other and are mixed to illustrate key, but different, characteristics of the data.

#### 3.1 Differences in SOLO competences between Computer Science and Mathematics

We compare the departments of Computer Science, Natural Science, and Mathematics. Our data in Natural Science consist of a compilation of the data of the departments of Physics, Biology, Chemistry, and Molecular Biology. We excluded Geology since it is a department only at AU. These four departments seem to be quite similar with respect to their average SOLO levels

(Brabrand & Dahl, 2008). Even though we basically want to compare Mathematics and Computer Science, we felt we needed a third partner in the comparison to shed light on some issues related to the two subjects, hence the choice of Natural Science as a comparison partner. In the following we use the terminology of calling the combined data set of the four natural science departments ‘the Natural Science department’ even though it is in fact several departments.

Looking at the SOLO averages at the two universities (Figure 13) we see that although there are some differences between the departments, they appear in the same order of SOLO hierarchy at each institution, and so does the average across both institutions. Furthermore the difference between ‘sister-departments’ at different universities (from 0.1 to 0.3) is generally smaller than the intra-university difference between departments (from 0.1 to 0.6). This indicates that it does make sense to pool the data from, for example, the two mathematics departments, and expect some meaningful data and conclusions.

Subject	AU	SDU	avg.	diff.
Computer Science	3.7	3.4	3.6	0.3
Natural Sciences	3.4	3.3	3.4	0.1
Mathematics	3.1	2.8	3.0	0.2
Intra-university diff.	0.3-0.6	0.1-0.6	0.2-0.6	

Figure 13: SOLO averages by department and university

Turning to the relative distribution of SOLO competences (Figure 14), we get a more detailed view of the

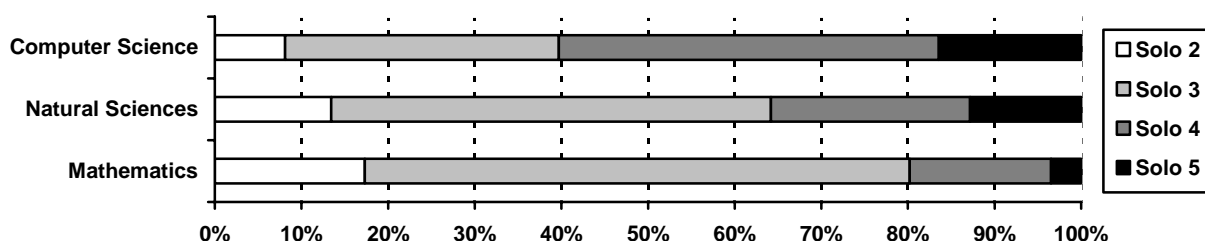


Figure 14: Distribution of competences by SOLO levels for the different departments at AU

TOP 10 COMPETENCES									
	Computer Science			Natural Sciences			Mathematics		
	Competence	SOLO	Freq.	Competence	SOLO	Freq.	Competence	SOLO	Freq.
1.	describe	3	13 %	describe	3	15 %	apply method	3	20 %
2.	explain	4	10 %	account for	3	13 %	reproduce	2	13 %
3.	apply method	3	9 %	apply method	3	9 %	solve	3	8 %
4.	implement	4	7 %	explain	4	8 %	formulate	3	8 %
5.	analyze	4	6 %	analyze	4	5 %	prove	3	7 %
6.	discuss	5	5 %	discuss	5	4 %	argue	4	5 %
7.	design	4	4 %	execute proc.	3	3 %	compare	4	5 %
8.	compare	4	3 %	identify	2	3 %	account for	3	5 %
9.	evaluate	5	3 %	assess	5	3 %	illustrate	3	3 %
10.	identify	2	3 %	formulate	3	2 %	combine	4	3 %
"Σ"			62 %			67 %			75 %

Figure 15: Top 10 competences for Computer Science vs. Mathematics vs. Natural Sciences

distribution of SOLO competences and what is behind the differences between the SOLO averages of the three departments. We see that the relative number of SOLO 2 and 3 competences *increases* and that the number of SOLO 4 and 5 competences *decreases* as we move down the departments. In other words, Computer Science uses more higher-level competences than Natural Science, which in turn uses more than Mathematics. Also, the majority of Computer Science competencies are qualitative competences (60% are at SOLO levels 4 and 5), while Mathematics and Natural Science both seem to use mainly quantitative competences (at SOLO levels 2 and 3).

To go even deeper into the differences between the departments, we investigated whether or not there is also a difference in the specific competences employed at the different departments. Figure 15 lists the 10 most-used competences for each of the three departments, an indication of the SOLO level to which they belong, and their frequency.

In Figure 15, we see that for Mathematics, the top 10 competences account for 75% of the competences employed, while the figure is substantially lower for Natural Science (67%) and Computer Science (62%). Mathematics therefore appears to be different from the other two departments since it seems that it uses a slimmer span of competences. Furthermore, SOLO 5 level competences are not part of the top 10 at the

Mathematics department whereas two of the competences in the two other departments are at SOLO level 5. Hence Mathematics seems generally to be more careful with using SOLO 5 competences.

Taking the top six of each department we get a list of 12 competences: 'describe', 'explain', 'apply method', 'implement', 'analyze', 'discuss', 'account for', 'reproduce', 'solve', 'formulate', 'prove', and 'argue'. If we compare their frequencies (in percent), we get the picture seen in Figure 16.

We see here that Mathematics again seems to separate out from the two other departments, which seem more alike than different. Particularly regarding the competences 'reproduce', 'formulate', 'prove', 'solve', 'apply method', and 'argue', Mathematics has at least twice as high a frequency as the two other departments. In relation to the competences 'describe', 'explain', 'analyze', and 'discuss', Mathematics has a frequency less than half that of the other departments. Only in the competences 'account for' and 'implement' does Mathematics seem less distinct from, and indeed to lie between the other two departments. It is only in 'implement' that Computer Science seems to be clearly separated from the other two departments. It seems that certain competences are much more common in some departments than others. We therefore grouped clusters of the competences that had separated out to see how much of the total amount of competences within a department they jointly accounted

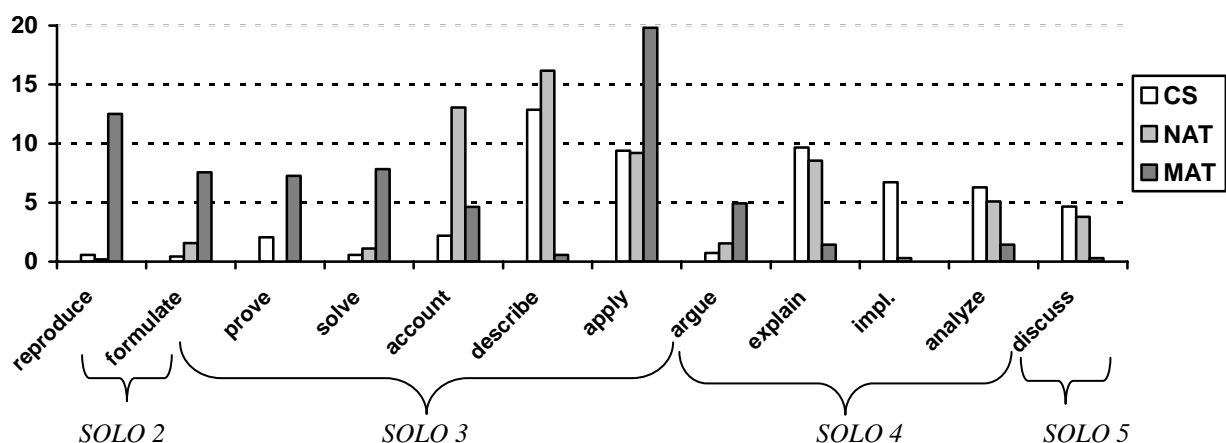


Figure 16: Freq. of common Computer Science competences compared to that of Mathematics and Natural Sciences.

Competence group:	Programming-related competences:	Competences where Mathematics is at 2x frequency (or more):	...and without 'apply':
Competences:	- implement SOLO 4 - program SOLO 4 - design SOLO 4 - construct SOLO 4 - structure SOLO 4	- reproduce SOLO 2 - formulate SOLO 3 - prove SOLO 3 - solve SOLO 3 - apply SOLO 3 - argue SOLO 4	- reproduce SOLO 2 - formulate SOLO 3 - prove SOLO 3 - solve SOLO 3 - argue SOLO 4
Computer Science:	15%	14%	4.5%
Natural Sciences:	1.0%	14%	4.4%
Mathematics:	0.3%	60%	40%

Figure 17: Competence clusters ('programming' and 'math-manipulation') by departments

for. This was to determine if some competences, regardless of SOLO level, were department-specific.

Grouping sets of competences related according to certain criteria provides interesting information, as shown in Figure 17.

#### Computer Science

Many people equate Computer Science with 'programming computers', but programming-related competences (e.g., 'program' 'implement', 'design', 'construct', and 'structure') occupy only 15% of the Computer Science curriculum (Figure 17). The same set of competences are negligible on the Natural Science curriculum, at 1.0%, and virtually non-existent on the Mathematics curriculum, at 0.3%. As expected, we thus see Computer Science standing out in this respect: we see that although programming is definitely an essential part of Computer Science – hence its reputation – it is by no means the main part.

#### Mathematics

Combining the competences where Mathematics distinctively separated itself out (i.e., 'reproduce', 'formulate', 'prove', 'solve', 'apply (method)', and 'argue') we see that together they account for 60% of all competences on the Mathematics curriculum; yet the same competences are remarkably less dominant in the other departments, with 14% for both Computer Science and Natural Science. In this respect, Mathematics seems to be distinctively different and one might argue that mathematics (at least the teaching of university mathematics) is to a large extent about reproducing, formulating, proving, solving, applying methods, and arguing. It so happens that according to the SOLO Terminology, these competences are 'lower' level, which is not to indicate that they are easy, nor that mathematics is in some sense 'lower' than the other subjects. If we remove 'apply', which often occurs in the other departments without directly involving mathematics, the difference ratio increases from about 1 : 4 (i.e., 60 : 14%) to about 1 : 9.

#### Natural Science

For Natural Science we tried to identify 'laboratory skills', but were unable at present to clearly isolate them based on competences alone. The competences 'execute

(procedure)' and 'carry out (instructions)', which may be somewhat related to laboratory work, come to 4.2% for Natural Science, 2.5% for Computer Science, and 1.7% for Mathematics. We were not able to come up with a logically related set of competences that strictly isolated Natural Science from Mathematics and Computer Science.

## 4 Conclusion

#### Different departments

All three methods of analyzing the data suggest the same conclusion, namely that the three departments are different in their use of SOLO competences. The overall SOLO averages seem quite different for Mathematics and Computer Science, with Natural Science as a comparison partner also being different from the other two. Most of the time, Mathematics stands distinct from Computer Science and Natural Science, which, although different, seem to be closer to each other than to Mathematics. One might wonder what the reason is for the difference between Mathematics and the other Natural Science subjects, including Computer Science. One reason might be that Mathematics is usually considered to be a *vertical* discipline, with a hierarchy of theories and methods building upon each other, whereas many other disciplines are *horizontal*, with theories and methods living side by side (Madsen & Winsløw 2007). Furthermore, one can argue that we are researching the *SOLO* competences and the SOLO model might not be as appropriate a tool for describing *mathematical* competences as for other subjects. The SOLO Taxonomy was created using all higher education subjects, not *only* mathematics. In fact extensive work has been done within mathematics education to describe what distinguishes *mathematical* competences; for instance, problem-solving, reasoning and proving, communicating, connecting, representing etc. (NCTM 2000; Niss 2002; PISA 1999).

#### Different competences (and SOLO distributions):

The three disciplines show differences not only at the level of averages but also in the usage and distribution of verbs. Mathematics seems to use lower SOLO levels, Computer Science more higher-level verbs, and Natural Science in the middle. Also, in terms of the distribution of the specific verbs, Mathematics seems quite different

from the two other departments. Computer Science stands out with its inclusion of programming-related competences and the fact that the majority of the competences are qualitative, while the majority of the Mathematics and Natural Science competences are quantitative. All this suggests that Mathematics and Computer Science are in fact the most 'different' departments of the three. This might seem remarkable given the mathematical nature and basis of Computer Science and that Computer Science 'grew out of' and is part of the Mathematics department at many universities. What might the reason be? It needs further study to answer, but as stated above, the characteristic competences at each of the three departments are placed by the SOLO model at different levels. Does this then *finally and ultimately* explain the core of the subjects? No, it is to some extent also a reflection of the different teaching (and cultural) traditions in the three departments. Furthermore, as stated above, one could also argue that the SOLO taxonomy might not fit each department equally well.

#### *SOLO and Constructive Alignment*

Although the SOLO taxonomy might not fit each department equally well, it is good tool to help create a discussion about the purposes of a course and the formulation of ILOs. It is also a helpful tool to point to areas of interest when analysing various syllabi and departments. Also, the process at the two university faculties of implementing ILOs based on the principles of constructive alignment and the SOLO Taxonomy have given us a big standardized dataset and the opportunity to investigate the syllabi. Something that was not possible before, partly due to the 'private' manner in which syllabi were written – usually each teacher with his own personal style. A note of caution is needed, however. Sometimes there are practical reasons, such as time or fiscal constraint, room availability etc., that constrain a teacher to hold exams in less than ideal circumstances. Taking the example from above, it could be that the teacher is forced to use the cheaper MCQ test instead of an oral exam, which he would have preferred owing to his ideal ILOs that mention competences such as 'construct' and 'formulate'. In that case, aligning his course to the actual MCQ test would force him to rewrite his ILOs to a much lower SOLO level, i.e. less ambitious and relevant. Should he actually do that – in the name of alignment? The question is difficult, but in these circumstances it might be better if he keeps his higher SOLO level ILOs, align his teaching to these ILOs even though Susan's behaviour is then not fully rewarded at the exam. She will get her reward later.

#### *'Creating' a language for competences (over time)*

One of the purposes in implementing the same model at both faculties is that it can help create a common language to be used among university teachers, since often any "speech community is likely to be composed of different groups, groups which may operate with differing versions of the same language or even with discrete and separate language" (Montgomery 1992, p. 101). Also, the syllabi prior to adopting the SOLO Taxonomy were for the most part not formulated using measurable operational competences. Finally, all these things might

ultimately synergize to improve learning since students would also, over time, be used to reading the operational syllabi, hence be more aware of what is expected, and hence be more inclined to act like Susan rather than Robert.

## 5 Acknowledgements

The authors would very much like to thank Anne Mette Mørcke, Berit Eika, Gitte Wichmann-Hansen, John Biggs, and Catherine Tang for providing extensive and valuable feedback on our SOLO classification of verbs. Also thanks to Torben K. Jensen and Gunnar Handal for interesting discussions about our analysis and the background for it.

## 6 References

- Bauersfeld, H. (1979). Research related to the mathematical learning process. (In International Commission on Mathematics Instruction, ICMI (Ed.), *New Trends in Mathematics Teaching, Vol. IV* pp. 199-213. Paris: UNESCO)
- Biggs, J. B. & Collis, K. F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy, Structure of the Observed Learning Outcome*. (London: Academic Press)
- Biggs, J. B. (2003). *Teaching for Quality Learning at University*. (Maidenhead: Open University Press)
- Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. (2003). *XQuery 1.0: An XML Query Language*. [Working Draft: <http://www.w3.org/TR/xquery/>]
- Brabrand, C. (2007). *Constructive Alignment for Teaching Model-Based Design for Concurrency*. Proc. 2nd Workshop on Teaching Concurrency (TeaConc'07), Siedlce, Poland
- Brabrand, C. & Andersen, J. (2007). *Teaching Teaching & Understanding Understanding*. 20 minutes award-winning short-film on Constructive Alignment (available on DVD from [www.unipress.dk](http://www.unipress.dk)), [homepage: [www.daimi.au.dk/~brabrand/short-film/](http://www.daimi.au.dk/~brabrand/short-film/)], Aarhus University Press
- Brabrand, C. & Dahl, B. (2008). Using the SOLO-Taxonomy to Analyze Competence Progression of University Science Curricula. Paper submitted to international journal on 15 January 2008.
- Goodlad, J. I., Klein, M. F., Tye, K. A. (1986). The Domains of Curriculum and Their Study. (In B. B. Gundem (Ed.), *Kompendium 3 Om læreplanpraksis og læreplanteori* (pp. 31-64). Pedagogisk Forskningsinstitut, University of Oslo) (Reprint of parts of Goodlad et al. (1979) book: Curriculum Inquiry. New York: McGraw-Hill Book Company)
- Madsen, L. M. & Winsløw, C. (2007). Relations between Teaching and Research in Physical Geography and Mathematics at Research Intensive Universities. *Manuscript, accepted modulo revisions for*

International Journal of Science and Mathematics  
Education

- Montgomery, M. (1992). *An Introduction to Language and Society*. (London: Routledge)
- NCTM (2000). *Principles and Standards for School Mathematics*. National Council of Teachers of Mathematics: VA, USA.  
<http://www.nctm.org/standards/default.aspx?id=58>
- Niss, M (2002). Kompetencer og matematiklæring – Ideer og inspiration til udvikling af matematikundervisningen i Danmark. *Temahæfteserie nr. 18*. Copenhagen: Ministry of Education  
<http://pub.uvm.dk/2002/kom/>
- Oliver, D., Dobeles, T., Greber, M., Roberts, T. (2004). This Course Has A Bloom Rating Of 3.9. *Australian Computer Society, Inc. Proceedings of the Sixth Australasian Computing Education Conference (ACE2004), Dunedin, NZ. January 2004 Conferences in Research and Practice in Information Technology, Vol. 30, 227-31. Editors Raymond Lister and Alison Young.*
- Oppenheim, A. N. (1992). *Questionnaire Design, Interviewing and Attitude Measurement*. (London: Pinter)
- PISA (1999). *Measuring Student Knowledge and Skills*. OECD Programme for International Students Assessment, Paris: OECD. pp. 41-57
- Robson, C. (2002). *Real World Research*. (Oxford: Blackwell)

## Appendix A: The data material

The data for the entire analysis is available online (in browsable XML format) here, in Danish:

Online location (URL):	Data	Description
<a href="http://www.itu.dk/people/brabrand/solo.xml">www.itu.dk/people/brabrand/solo.xml</a>	SOLO data	SOLO attribution of all competences occurring in both data sets (cf. below).
<a href="http://www.itu.dk/people/brabrand/data-au.xml">www.itu.dk/people/brabrand/data-au.xml</a>	AU data	All competences for all ILOs for all courses at all NAT/AU departments.
<a href="http://www.itu.dk/people/brabrand/data-sdu.xml">www.itu.dk/people/brabrand/data-sdu.xml</a>	SDU data	All competences for all ILOs for all courses at all NAT/SDU departments.

The data is represented in the following XML format:

<pre> &lt;!ELEMENT xml (institute*)&gt; &lt;!ELEMENT institute (group*)&gt; &lt;!ELEMENT group (course*)&gt; &lt;!ELEMENT course (goal*)&gt; &lt;!ELEMENT goal (competence*)&gt;  &lt;!--   !ATTLIST institute name CDATA #REQUIRED   !ATTLIST group name CDATA #REQUIRED   !ATTLIST group season CDATA #REQUIRED   !ATTLIST group year CDATA #REQUIRED   !ATTLIST course name CDATA #REQUIRED   !ATTLIST course ects CDATA #REQUIRED   !ATTLIST course id CDATA #REQUIRED   !ATTLIST goal value CDATA #REQUIRED   !ATTLIST competence value CDATA #REQUIRED --&gt; </pre>	<pre> &lt;xml&gt; &lt;institute name="Computer Science"&gt; &lt;group level="Undergrad" season="Fall" year="2007"&gt;   &lt;course name="Algorithms and Datastructures II"     ects="5" id="7819"&gt;     &lt;goal value="construct and analyze algorithms       using standard algorithm paradigms"&gt;       &lt;competence value="construct" /&gt;       &lt;competence value="analyze" /&gt;     &lt;/goal&gt;     &lt;goal value="identify and formulate algorithmic       problems as graph and string       problems"&gt;       &lt;competence value="identify" /&gt;       &lt;competence value="formulate" /&gt;     &lt;/goal&gt;   ... </pre>
--	--

Data representation (in DTD format)

Sample data fragment (translated from Danish)

The data can then be queried by, for instance, XQuery (Boag et al. 2003) programs such as the following which calculates the frequencies of all *SOLO level 4* competences from *Computer Science* courses at AU and outputs them in *descending* order of their frequency counts:

<pre> &lt;result&gt; { let \$competences := fn:doc("data-au.xml")//institute[@name eq "Computer Science"]//competence let \$verbs := \$competences/@value for \$verb in fn:distinct-values(\$verbs) let \$solo := fn:doc("solo.xml")//competence[@value = \$verb]/@solo let \$frequency := fn:count(\$all_competences[@value = \$verb]) where \$solo eq 4 order by \$frequency descending return &lt;competence value="{ \$verb }" solo="{ \$solo }" freq="{ \$frequency }"/&gt; } &lt;/result&gt; </pre>
---

## Appendix B: Sticky yellow notes

At the beginning of his keynote presentation, Claus Brabrand posed the question, "What is good teaching?". The audience members wrote the following answers ...

*... that helps students reach the learning goals that they/we have set.*

When you get the pupils in a way that they want to learn it.

**Good knowledge on your subject.  
Clearly present that to others.**

Understanding that teaching doesn't mean that students actually learn ...

*Something that reaches the student and makes him think.*

Activation, provocation; leads to new insights and motivation.

- Student-centred
- Teacher-student interaction
- Giving the student responsibility for the learning
- Active teaching-learning environment

*To share knowledge and to have contact with learners.*

*Inspiring a student to learn beyond what they are "taught".*

*... is getting learners to have the skills and concepts and attitude.*

*To not to teach but tutor or guide the learner through learning.*

**Seeing relevance to facts  
- Student outcome**

**Clear  
Organised  
Interactive**

Creating  
possibilities

*Good teaching is teaching  
that inspires good learning*

*Teaching that  
motivates students  
to want to learn*

Good teaching supports  
students' learning process  
so that it is faster,  
more complete

**Clear  
Inspiring**

*Creating an understanding in  
students with an appreciation of  
its value.*

**Good teaching happens when  
the teacher jumps into the  
head of students, understands  
their preconceptions.**

Good teaching facilitates "good"  
learning outcomes for as wide  
a range of learners as practicable.

*Is what helps others to learn;  
does more good than harm*



*Teaching that informs  
effectively by entertaining.*

- Teaching that engages
- Teaching that takes the  
perspective of the student  
and focuses on the subject

**What advances  
good learning**

*Good teaching is teaching  
that warrants good learning.*

- interactive
- motivating



---

Claus Brabrand gives his own answer to the question in his 19-minute award-winning short film (DVD) about Constructive Alignment, *“Teaching Teaching and Understanding Understanding”*. You can watch and order the film at:

<http://www.daimi.au.dk/~brabrand/short-film/>



# RESEARCH PAPERS



# What does “Objects-First” Mean?

## An International Study of Teachers’ Perceptions of Objects-First

**Jens Bennedsen**

IT University West  
Fuglsangs Allé 20  
DK-8210 Aarhus V  
Denmark  
jbb@it-vest.dk

**Carsten Schulte**

Department of Computer Science  
Free University  
Takustrasse 9  
D-14195 Berlin, Germany  
Schulte@inf.fu-berlin.de

### Abstract

In this article we study how teachers of objects-first versions of CS1 courses all over the world understand the term ‘objects-first’.

By analysing the descriptions of the term objects-first from more than 200 teachers worldwide, we have found and described three categories: using objects, creating classes, and concepts. A second study of more than 40 teachers was undertaken to validate our suggested descriptions.

Implications of the three categories are described. These were determined by analysing whether the three categories lead to different evaluations of typical CS1 topics. The differences are analysed in relation to three dimensions: the relevance of the topic, how difficult teachers think students find the topic and the level at which students are supposed to learn the topic. Overall, only small differences between the categories are found.

The three categories characterize a little over 50% of the courses; the rest use a combination of the three categories.

**Keywords:** Objects-first, CS1, object-oriented programming, curriculum design.

### 1 Introduction

In Computing Curriculum 2001 (Engel and Roberts 2001), three different implementations of a programming-first CS1 course were presented: imperative-first, objects-first (OF in the following) and functional-first. OF is only vaguely defined in Computing Curriculum 2001: “The objects-first model also focuses on programming, but emphasizes the principles of object-oriented [OO] programming and design from the very beginning ... The first course in either sequence begins with the notion of objects and inheritance, giving students early exposure to these ideas. After experimenting with these ideas in the context of simple interactive programs, the course then goes on to introduce more traditional control structures,

but always in the context of an overarching focus on OO design” (p. 30).

Since the publication of the Computing Curriculum in 2001 there has been a lively discussion on the idea of OF. Back in March 2004, a dynamic discussion took place on the SIGCSE mailing list (SIGCSE-members 2005). For a summary of the discussion, see Bruce (2005). The debate has been the basis for an ITiCSE working group applying “four distinct research methods to the discussion: cognitive science, rhetorical analysis in the critical tradition, phenomenography and biography” (Lister et al., 2006, p. 146). In fall 2006, Michael Kölling reopened the debate with his “I Object” contribution sent out on the SIGCSE mailing list. Panels at conferences have debated whether one should use an OF approach when teaching introductory programming (Astrachan et al. 2005, Bailie et al. 2003), and teachers have written many reports on their implementations of OF (e.g. Bruce, Danyluk and Murtagh 2001, Cooper, Dann and Pausch 2003, Hu 2004, Moritz et al. 2005), but it still seems that a common understanding of the term is missing. However, as Lewis (2000) points out, it is “often used to convey the general idea that objects are discussed early in the course and established as a fundamental concept. Beyond that, however, these phrases seem to take on a variety of meanings, with important implications” (p.246).

This discussion led us to the following research questions:

1. What are the different understandings of the term ‘objects-first’ among introductory programming teachers?
2. What are the ‘important implications’ of these different understandings of the term ‘objects-first’ for teaching an introductory programming course?

### 2 The Research Design

In this section we argue for the research design chosen.

#### 2.1 The Overall Design

The research questions deal with the hypothesis that teachers have different understandings of OF. It is a study aimed at building new categories representing the general views of teachers. A study design seemed appropriate where teachers could describe their understandings of the term. Since the respondents are from all over the world,

an electronic distribution would be required. The response rate among computer science teachers was expected to be higher using this form than using a paper and pencil version. To make the responses manageable, the teachers were asked for a description of their understanding of OF.

The descriptions were used as the basis for content analysis (Stemler 2001) in order to generate categories of understandings of the term. The purpose of content analysis is “compressing many words of text into fewer content categories” (Stemler 2001). A category is “a group of words with similar meaning or connotations” (Weber, 1990, p. 37)

The second research question focuses on ‘important implications’. How do we define and measure important implications? The collection of data for this article was part of a larger study on teaching topics in introductory programming (Schulte and Bennedsen 2006). This larger study focused on three aspects of different topics in introductory programming: relevance of a given topic (referred to as *relevance*), the learning difficulty of a given topic (referred to as *difficulty*) and the desired cognitive level a given topic should be learned at (referred to as *level*).

Is it possible to describe general variations between the different categories? In order to answer this question, an analysis of statistically significant differences between the difficulty, relevance and level of a list of typical topics from introductory programming was undertaken. As described in Schulte and Bennedsen (2006), the list of topics was compiled from Dale (n.d.a), Dale (n.d.b), Engel and Roberts (2001) and Milne and Rowe (2002).

The data collection was done using a web-based questionnaire, which can be found at Bennedsen and Schulte (n.d.)

## 2.2 The Population

The biggest problem is which teachers to address. The research question mentions “introductory programming teachers”. This is a large group of people, and there is no database from which a random selection of of them can be drawn. An accessible group of teachers is the group attending appropriate conferences. While attendance lists for conferences are traditionally not accessible, the authors of articles for that conference are listed in the proceedings. In this way it is possible to address teachers at university and college level. Another appropriate group is high school teachers. The following six groups of teachers were chosen:

- The authors of articles for Koli Calling 2004, the 4th Annual Finnish / Baltic Sea Conference on Computer Science Education (Korhonen and Malmi 2004);
- The authors of articles for, and the participants in panels at, the 36<sup>th</sup> Technical Symposium on Computer Science Education (SIGCSE’05 2005);
- The authors of articles for, and panel members at, the 10<sup>th</sup> Annual Conference on Innovation and

Technology in Computer Science Education (ITiCSE’05 2005);

- The authors of research papers at the 4<sup>th</sup> International Conference on Advanced Learning Technologies (ICALT’04 2004);
- The authors of articles at the Australasian Computing Education Conference (ACE’04 2004);
- The ACM (Association for Computing Machinery) Special Interest Group on Computer Science Education’s email list (SIGCSE mail-list n.d.);
- The teachers at the Danish Advanced Computer Studies program (DMLF 2006)
- The K  nigstein group (a group of computer science teachers in German high schools) (Koenigstein n.d.)

The reason for choosing these groups is that they consist of teachers seemingly interested in computer science education (SIGCSE, ITiCSE, Koli, ACE and the SIGCSE email list), and of teachers interested in teaching technology but not necessarily interested in computer science education (ICALT). The four computer science education conferences (SIGCSE, ITiCSE, Koli and ACE) attract teachers from all over the world. SIGCSE is held in the United States, ITiCSE in Europe and ACE in Australia or New Zealand, while Koli is held in Finland and attracts teachers mostly from the Baltic Sea area. ICALT participants are from all over the world. The SIGCSE mailing list was used to address teachers who were not necessarily authors, but who still had an interest in computer science education. The teachers at the Danish Advanced Computer Studies program teach at college level. The K  nigstein group focuses on high school teachers.

It is debatable whether the selected groups are representative of introductory programming teachers. It might be argued that teachers who participate in conferences focused on teaching have a stronger interest in the field than those who do not. It is important to bear this in mind, because the numbers presented in the following sections are likely to lack a degree of preciseness, due to problems in measuring the representativeness of the people asked.

According to the International Association of Universities (International Handbook of Universities 2005), there are approximately 9,000 universities (not including colleges) all over the world. Added to that is a large number of colleges (in the US alone, there are almost 7,000 universities and colleges (National Centre for Educational Statistics 2007)). Unfortunately, we have no numbers of the percentage of universities with a computer science program world wide, but in the US 649 institutions offer a bachelors or masters degree in computer science or mathematics and computer science (National Centre for Education Statistics 2007b), i.e. 10% of the universities and colleges offer a computer science degree. If each university and college with a degree in computer science has 20 members of faculty, the total population of

computer science university teachers is approximately 18,000. How large should the sample size be? This depends on the level of precision, the confidence level (if a 95% confidence level is selected, 95 out of 100 samples will have the true population value within the range of precision) and variability (the more heterogeneous a population, the larger the sample size required to obtain a given level of precision. Yamene (1973) provides a formula to calculate the sample size (given variability of 0.5 and a confidence level of 0.95):

$$n = \frac{N}{1 + N * e^2} = \frac{18000}{1 + 18000 * 0.05^2} = 391$$

where  $n$  is the sample size,  $N$  is the size of the population and  $e$  is the level of precision

The questionnaire was sent to 771 named teachers as well as the recipients of the SIGCSE mailing list (SIGCSE mail-list n.d.). Furthermore, we asked the recipients to pass on the invitation to other interested teachers. Of the 771 named invitations, 71 email addresses were bounced, giving 700 named respondents. Overall, 457 teachers viewed the questionnaire; 298 answered the parts of the questionnaire related to this study, with 238 completing the full questionnaire in an average of 19 minutes. The response rate (percentage of named respondents who answered the questions for this study) was 34 %.

The precision can be calculated using the formula above; in our case it is 0.064 (if for example our analysis says that 35% of the respondents teach in a certain way; we can conclude that the real number is in the range 28.6% to 41.6%). The precision is not as high as expected but still high enough to give general trends for the population.

### 3 Understandings of Objects-First

A content analysis of the answers was performed in the following way:

1. One of the authors read through all the answers and formed four categories. These four categories were described and the answers were placed in the categories.
2. The second author read through the answers and categorized them based on the first author's description of the categories. For the first three groups there was substantial agreement. In the last group, there was both disagreement and a few observations. The fourth category was eliminated.
3. The few answers categorized differently by the two authors were discussed and used to form a clearer definition of the three categories.

We used an emergent coding. Stemler (2001), drawing on Haney, Russell, Gulek, & Fierros (1998), describes the procedure for this type of content analysis as the following:

"First, two people independently review the material and come up with a set of features that form a checklist. Second, the researchers compare notes and reconcile any differences that show up on their initial checklists. Third, the researchers use a consolidated checklist to

independently apply coding. Fourth, the researchers check the reliability of the coding (a 95% agreement is suggested; .8 for Cohen's kappa). If the level of reliability is not acceptable, then the researchers repeat the previous steps. Once the reliability has been established, the coding is applied on a large-scale basis."

The content analysis led to the suggestion of the following three categories (the descriptions of the categories were adjusted later based on comments from teachers):

1. **Using objects:** At the beginning of the course, the student uses objects implemented beforehand. When the student has understood the concept *object*, he moves on to defining classes by himself. Focus is on usage before implementation.
2. **Creating classes:** The student both defines and implements classes and creates instances of the defined classes. Focus is on the concrete-creative part of programming.
3. **Concepts:** This involves the teaching of the general principles and ideas of the object-oriented paradigm, focusing not just on programming but on creating object-oriented models in general. Focus is on the conceptual aspects of object-orientation.

In order to offer a better understanding of the three categories, some prototypical examples of answers are provided:

1. **Using objects:** "using Objects before programming classes (and Objects before Algorithms)"; "Objects interaction introduced before methods implementation".
2. **Creating Classes:** "Writing and using classes before working with algorithms"; "The students are working with the concepts of classes and objects, instantiation, data fields and methods from day one"; "Teaching objects and classes from the very beginning of the course. (Maybe the way I teach is more like classes first?)".
3. **Concepts:** "Introduce the concept of objects before delving into programming concepts"; "Learn to see the world as objects and to map these objects to classes of code".

The above categorizations can be seen as an extension of the two different views of OF described by Lewis (2000): "A distinction must quickly be made between initially *writing classes* that define objects, and *using objects* defined by pre-existing classes.... Most educators agree, however, that using objects is the appropriate first step" (p. 246).

Thirty two percent (75 in total) of the descriptions were too short to categorize. Moreover, many of the descriptions defined OF as "something else" or in opposition to procedural first, indicating a focus on OO but not necessarily an OF approach: "Starting with object-oriented concepts rather than procedural concepts." Fifty fell in the category 'other.' We found that most respondents (51) use a *creating-classes*

approach, followed by 34 who use *concepts* and 28 *using objects*.

### 3.1 Validation of the Three Categories

For further validation of our findings, a second study was done. A short web-based questionnaire was sent to the 69 teachers who had indicated in the first questionnaire that they used an OF approach in their introductory programming course, and had allowed us to contact them again. Three were not reachable by email. Forty completed the short questionnaire.

The teachers were asked if they found the descriptions understandable and how they would describe their course in terms of the categories suggested.

All respondents found the descriptions of the three categories both useful and understandable. Some improvements for the descriptions were suggested, which have now been incorporated into the descriptions above. The original phrasing can be found in the questionnaire at Bennedsen and Schulte (n.d.).

We asked the teachers to comment on each of the suggested categories. Below we briefly summarize these open-text comments (the numbers exclude comments like “ok” or equivalent; quotes are in *italics*):

1. **Using objects** (10 comments): The comments focused on the word ‘using’ and the many concrete understandings of that word: *includes writing code or not? Includes instantiating objects with a tool like blueJ? Looking at and understanding the class description, how it is built and what instance variables and methods means? Explicitly, by means of actual programming, i.e., code that instantiates and uses objects. Implicitly, by means of some kind of tool (games, animations ...) that takes care of actually instantiating the objects.*

2. **Creating classes** (18 comments): Four comments were on pedagogical implications like: *more difficult / more easy for students, needs a lot of syntax*. Two comments were on the necessity for students to understand good OO design: *explicitly excluding the idea that writing one class with one main method and no other methods is not defining and implementing their own classes; should include ‘identifying’ classes*. Two did not understand the term ‘concrete-creative’. One found the distinction between object (first category) and class (this category) unclear.

3. **Concepts** (14 comments): Most of the comments were on pedagogical implications: *I try to introduce concepts, such as inheritance, in informal ways, without “constructive orientation” we experienced that students easily model nonsense*. Others asked for a more explicit concept list (one wanted to include polymorphism while another wanted to exclude this topic). Lastly, a distinction from creating classes was suggested: *Be more explicit that programming is de-emphasized in favour of building models and using pseudo-code Clarify distinction from classes first by including topics like using UML, clarify the role of programming*

According to Dictionary.com (n.d.), *use* means “to employ for some purpose; put into service; make use of.” In the definition of *using objects*, the role of classes is therefore to put the objects into service. We have not described in the definition how to put the objects into service; this could be done using a tool like BlueJ (Kölling et al. 2003) or by writing statements in a method. In the latter case, the coding is seen as a means of instantiating objects, rather than as having a purpose in itself.

When creating classes, we assume that teachers focus on the quality of the code produced by the students; this is implicit in the description of the category.

Instead of or as well as commenting on our categories, many teachers either a) used the categories to reflect on their personal approaches and subsequently reported which category they found more important and which was their favourite, or b) used the categories to describe ‘phases’ for an introductory course and commented on which category was the best first phase followed by their second choice.

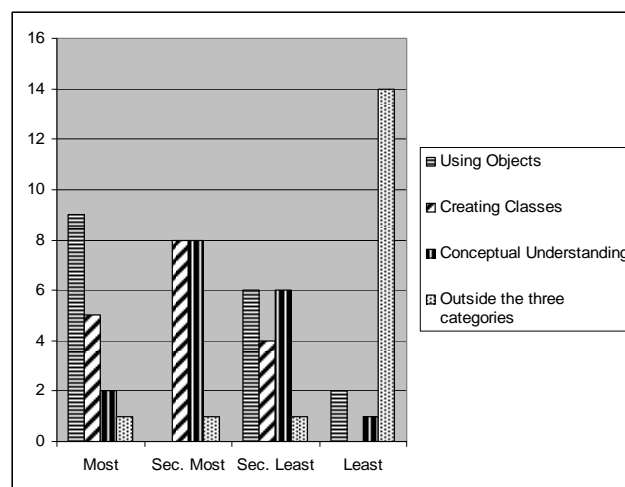
About half the respondents (21, or 53%) would characterize their introductory programming course as belonging to *exactly* one of the three categories.

When asked to rank the matching of the three categories to their programming course, the majority of the remaining respondents chose using objects first as the best match (figure 1). The second most popular category is either creating classes first or concepts. At the least prominent level, teachers rank “something else that is not covered by the three given categorizations.”

From figure 1 we can deduce three common sequences of OF courses:

1. Using objects, followed by
  - a. creating classes, followed by concepts.
  - b. concepts, followed by creating classes.
2. Creating classes, followed by concepts.

Note that no respondents answered using objects as the



**Figure 1: Teaching ingredients for objects first that do not follow exactly one of the suggested categories of OF**

Abbreviation	Topic
AdvDataStr	Advanced data-structures (linked-lists, trees, ...)
AlgDesign	Algorithm design (including designing the algorithms to implement the methods for a class)
AlgEfficiency	Algorithm efficiency (big-O)
CRC-cards	Responsibility driven design, CRC
Debugging	Debugging
DesignClasses	Design of classes (given a problem, determining the classes needed to solve the problem)
DesignSglClass	Design of a single class (number of constructors, private methods etc)
Encapsulation	Encapsulation (what should be private and what public?)
Ethics	Ethics
Generics	Generics (templates, type parameterization)
IDE	Using the program development environment (IDE, debuggers, and file organization)
Libraries	The libraries associated with the language
MentalModel	Mental model of the computer
MethodDesign	Design of methods (given the classes needed to solve a problem, specifying the methods)
Obj&Class	The concept of objects and classes
ObjComm	Object communication
Parameters	Parameters (what is the correct type and number of parameters to a given function/method?)
Poly&Inheri	Polymorphism and inheritance
ProbSolStrat	Divide and conquer (decomposition of a problem), stepwise refinement or other problem solving strategies
Ptr&Refs	Pointers and references
Recursion	Recursion
Scope	Scope of variables
Sel&Iter	Selection and Iteration
SimpDataStr	Simple data structures (arrays, strings, ...)
StatVsNonStat	Static vs. non-static (methods and variables)
Syntax	The programming language syntax
UMLClassDiag	UML class diagrams (reading them)
VarTypes	Instance and other type of variables

**Table 1: List of topics in an introductory programming course**

second most important category!

After this ranking of categories, we asked for additional comments in cases where the favoured approach of the teacher fell outside the given categories. We received 10 such comments. Interestingly, five of these described approaches that combined the given categories (and even used the names of the categories), such as: *I tend to use an approach that combines "Using objects first" with "Classes first" and adding a dash of philosophy [philosophy is our old term for the concepts category] or A mix: 1. short discussion on the philosophy, 2. glancing at and testing some predefined classes, 3. introduction to how classes are built in general, and then 4. more details*

*of the syntax.* One comment described a spiral approach: *Our emphasis is on design, but every design is followed by an implementation and we are reusing some classes in later exercises.*

Overall, we conclude that the categories are understandable and provide a good description of the different approaches to OF. We further find that the general approach to OF is a mixture of the three categories, i.e. the categories can be seen as “ingredients” in the curriculum; therefore they provide a very useful vocabulary for describing and discussing different course designs.

In summary we found six variations which rely on the three main understandings of OF. These are:

1. using objects,
2. creating classes,
3. concepts,
4. using objects → concepts → creating classes,
5. using objects → creating classes → concepts, and
6. creating classes → concepts.

In the next section we will examine whether these variations lead to implications for teaching.

#### 4 Implications for Teaching

The second research question focuses on what Lewis (2000) called “important implications”: are there differences in teaching according to different understandings of the term OF?

In order to analyse this question, the respondents were grouped according to their understandings of the term OF. We used a subgroup of the respondents of Schulte and Bennedsen (2006); respondents who

1. followed an OF approach,
2. explained their understanding of OF, and
3. have experience in teaching such courses.

This subgroup consists of 110 persons. They were grouped into four categories ‘use objects’, ‘create classes’, ‘concepts’ and ‘other’.

The teachers rated 28 topics of a typical introductory programming course (see table 1) from three different perspectives: the **difficulty** of each topic (how difficult the teacher finds the topic for the student), the **relevance** (how relevant the teacher finds the topic in an introductory programming course) and the intended **level** of understanding that the teacher thinks the student should have when the course is finished, according to Bloom’s taxonomy. Each dimension was rated on a five-point scale; the low number indicates that the teacher thinks the topic is easy/not very relevant/should be learned at a low level. For details, see (Schulte and Bennedsen 2006). We checked for differences between the four groups with respect to the rating of the topics. In the following, we present the results of the ANOVA tests (Wallnau and Gravetter, 2005) for the three categories.

Tables 2, 3 and 4 list topics with significant differences between the teachers in the three categories.

Level	OF type				ANOVA	
	Use Objects	Create Classes	Concepts	Other	F	p
IDE	3.96	3.08	3.79	3.30	3.25	.03

**Table 2: Level <> categories of OF, one-way ANOVA**

Difficulty	OF category				ANOVA	
	Use Objects	Create Classes	Concepts	Other	F	p
StatNonStat	3.25	3.62	3.59	2.65	6.71	.00
VarTypes	2.35	2.96	2.67	2.38	4.14	.01
Ethics	2.13	1.64	2.89	2.24	2.86	.05

**Table 3: Difficulty <> categories of OF, one-way ANOVA**

Relevance	OF category				ANOVA	
	Use Objects	Create Classes	Concepts	Other	F	p
Sel&Iter	4.96	4.79	4.42	4.34	3.77	.01
AdvDataStr	2.23	2.00	2.63	2.83	2.83	.04
Scope	4.57	4.57	3.86	4.26	2.90	.04
MentalModel	3.00	3.88	3.68	3.41	3.03	.03
VarTypes	4.52	4.48	3.77	4.07	3.11	.03
Libraries	3.87	4.07	3.33	3.32	3.34	.02

**Table 4: Relevance <> categories of OF, one-way ANOVA**

## 4.1 Interpretation of Findings

In this section we try to find explanations for the differences we found. We therefore interpret the findings with the above given characterizations of the different understandings of OF.

### 4.1.1 Level of Understanding

The different categories have no or very small effects on the intended level of understanding (table 2). The only significant difference is found regarding the use of an IDE. This result fits the above described comments of teachers in the second study: if one aims to use objects without writing classes, other means are needed, e.g. an IDE such as BlueJ. In particular, students have to use more challenging functions of the IDE, perhaps including debuggers to use objects. In contrast, when using the approach 'create classes', students focus more on the mechanics of programming and therefore do not need to use sophisticated functions of an IDE; accordingly, in this group the level of using an IDE has the lowest value.

Interestingly, the concepts group also places a high value on using an IDE. This indicates that teaching conceptual understanding first includes similar pedagogical use of an IDE to demonstrate or explore concepts.

A further interpretation is that differences in OF approaches result in a different pedagogical role of the IDE as a learning tool.

### 4.1.2 Difficulties of Topics

The perceived difficulties of topics (table 3) show few statistically significant differences.

One implication of teaching one of the three 'pure' approaches is that students face more difficulties understanding the differences between static and non-static aspects, compared to those taught by an 'other' approach. In the discussion above we have seen that 'other' approaches generally consist of mixes of two or three of the 'pure' approaches. This observation gives a possible explanation of the difference: the 'pure' approaches focus on either static or non-static issues (e.g. it could be that creating classes focuses more on class (static) attributes than using objects), whereas a mix of approaches (the 'other' category) can help students to understand both static and non-static methods and attributes, and might consequently give rise to fewer difficulties.

Instance variables and other types of variable (VarTypes) are seen as more difficult by the 'create classes' group than by any other group. A possible explanation is that students using this approach create many more classes and consequently see more problems with this concept.

Similarly, a possible explanation for the highest difficulty of 'ethics' in the concepts group could be due to this approach teaching issues on a more conceptual level, making a topic such as ethics much more difficult.

Differences in OF approaches have some minor impact on learning difficulties. One interesting consequence of the interpretation of differences regarding "StatNonStat" is that learning CS topics can profit from presenting concepts from different perspectives. However, we are not sure whether this explanation applies to the differences in "VarTypes". Overall, we interpret the results in difficulties as a hint that the different versions of OF might have only a minor impact on learning difficulties.

### 4.1.3 Relevance of Topics

We find some differences regarding relevance (table 4). For six of the overall 28 topics the differences are significant.

'Selection and iteration' is more relevant for the groups 'use objects' and 'create classes', than for the 'other' group. A similar pattern is found for the topics 'scope of variables', 'instance variables' and 'using class libraries'. One interpretation is that 'using objects' and 'creating classes' versions of OF stress the importance of concrete examples and programming issues more than OF versions that focus on conceptual understanding. It fits this interpretation that the values of the 'other' group are consistently between use-objects and classes on the one side and concepts on the other side, because this group can often be seen as a mix of the others.

The reverse situation for the topic ‘advanced data structures’ fits this interpretation: advanced data structures are of only minor importance regarding concrete programming problems in beginners’ courses. However, they might be of greater importance for a more theoretical and conceptual understanding. Again, this is a hint that the ‘using objects’ and ‘creating classes’ versions of OF focus more on concrete programming experiences.

The differences of the relevance of the topic ‘mental model’ (‘teach students an appropriate understanding of how the computer executes an object oriented program’) do not fit this interpretation. This topic should be as relevant for approaches focusing on concrete programming as it is for conceptual approaches. However, while it is ranked high by the ‘create classes’ group, it is ranked quite low by the ‘using objects’ group – although above we found several hints that both groups focus on concrete programming. One possible explanation is that using objects helps students to develop a mental model without the need to explicitly teach it, while in other approaches one has to teach it explicitly. Of course, this interpretation would suggest that ‘mental model’ is highly relevant, but teachers of the ‘using objects’ group are not aware of this and give the topic a lower relevance than it should have. That the ‘create classes’ group ranks it higher could be due to these teachers focusing more on students’ own implementation of methods, in which case the students must have a better understanding of the semantics of executing the method. Such an understanding requires a good understanding of the memory model of the running program.

Overall, regarding relevance the groups ‘use objects’ and ‘create classes’ are quite closely connected, despite the notion of talking about using objects first *in contrast* to creating classes first. We explore this issue further in the next sub-section.

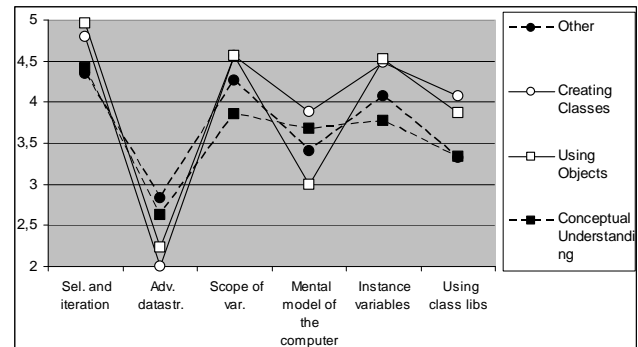
## 4.2 Summary of Interpretation and Conclusion

The impacts of the different versions of OF are not as great as we expected. Overall, there are only small differences. This indicates that OF proponents, despite different understandings of OF, have common goals and ideas about relevant contents of an introductory course. It seems that the “important implications” mentioned by Lewis (2000) are not related to different perceptions of the topics of an introductory programming course. This result is in sharp contrast to a comparison between OF and non-OF proponents, who differ strongly in intended level and perceived difficulty of topics (Schulte and Bennedsen 2006).

The difference between the three approaches is mostly a difference in the weighting of the importance of topics for an introductory course. Why are there differences regarding relevance (table 4)? Why do we find most differences in ‘relevance’, and hardly any in ‘level’ and ‘difficulty’? Do these findings match the conclusion of the previous section, where we suggested that the different understandings of OF were due to different ‘ingredients’ of an OF course? We think the finding

indicates that ‘relevance’ does not mean the relevance of the topic ‘in its own right’ (which is addressed in ‘level’), but rather its relevance for understanding other topics or for an overall understanding. In other words, the different understandings of OF go hand in hand with differences in teaching methods and the sequence of topics.

A closer look reveals that the two common approaches to OF – creating classes and using objects – are very closely related. Regarding relevance, they differ only for the topic *mental model* (see figure 2). They build a group (the OC group, white in figure 2), while the other two approaches (conceptual understanding and other) build another group (the CO group, black in figure 2).



**Figure 2: Comparison of OF types for differences in relevance of topics, scale: mean values of the groups, range: 1-5**

In the OC group, mean values range from 2 to nearly 5; in the CO group, the range is from nearly 3 to 4.5. The CO group estimates the relevance of ‘advanced data structures’ lower, while ‘using class libraries’ is rated much higher.

It may be that the different understandings of OF are due to organizing different ‘platforms of understanding’ for the students: using objects builds a platform that starts by learning the execution of an OO program. However, it remains somewhat fuzzy what other platforms are built by the other variants of OF.

## 4.3 Methodological Discussion

Our application of content analysis differs in some ways from the procedure proposed by Stemler (2001).

- He proposes that two authors read the texts independently and create categories. In our research, one author read the texts and created the categories. Following that the other author read the texts and checked the coding.
- We have not made explicit coding rules but implicit ones based on prototypical texts.
- We have not checked inter-rater reliability, since the coding was done in collaboration.

In this research, we checked the validity of the categories by asking the participants if the categories were understandable and asking them to use the categories to describe their own teaching.

It could be a mistake to use the description of the teachers' understandings of OF as a categorization of their teaching. Teachers might not teach according to their definition of OF.

Using teachers' understandings of students' problems could be misleading.

## 5 Conclusions

The debate on how to teach introductory programming has been intense. It is also blurred, since the different participants have different understandings of terms such as OF. In Computing Curriculum 2001, the description of the approach called OF is rather vague. By performing a content analysis of over 200 responses from teachers around the world, we have found three different categories of the concept: using objects, creating classes, and concepts.

The important implications of these three categories are not evident in the list of topics for an introductory programming course. Further research is needed to establish the implications. They could be in the use of different teaching methods or the sequence of topics taught.

It is our hope that this study will make the discussion on OF more precise and fruitful and thereby lead to new and innovative ways of teaching introductory programming. One interesting observation was that teachers immediately used the three categories to describe their own teaching, either using one category as a complete description of their course or using more than one as sub-sequences in the overall sequence. The use of these three categories seems to be a simple, useful and sufficient model to describe variations of OF, and therefore can foster a more precise pedagogical discussion on CS1 and the development of tools or exercises that support a particular way (category) of teaching introductory programming.

## 6 Acknowledgements

We would like to thank all of the respondents to our questionnaire; without them, this research would have been impossible. We would furthermore like to thank Michael Caspersen and Marianne Georgsen for constructive comments and discussions on earlier versions of this article.

## 7 References

- ACEC'04 (2004): Proceedings of the Australian Computers in Education Conference 2004.
- Astrachan, O., Bruce, K., Koffman, E., Kölling, M. and Reges, S. (2005): Resolved: objects early has failed. *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, St. Louis, MO, United States, 451.
- Bailie, F., Courtney, M., Murray, K., Schiaffino, R. & Tuohy, S. (2003): Objects first – does it work? *Journal of Computing in Small Colleges*, 19(2), 303-305.
- Bennedsen, J. and Schulte, C. (2006): Interaction hierarchy. <http://www.daimi.au.dk/~jbb/icer06.html>. Accessed 4 May 2006
- Bennedsen, J. and Schulte, C. (n.d.): *Objects-first survey*. [http://www.daimi.au.dk/~jbb/OF\\_survey.htm](http://www.daimi.au.dk/~jbb/OF_survey.htm). Accessed 28 August 2007.
- Bruce, K.B. (2005): Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)* 37(2): 111-117.
- Bruce, K.B., Danyluk, A. and Murtagh, T. (2001): Events and objects first: an innovative approach to teaching JAVA in CS 1. *CCSC '01: Proceedings of the sixth annual CCSC northeastern conference on computing in small colleges*. Middlebury, Vermont, United States.
- Cooper, S., Dann, W. and Pausch, R. (2003): Teaching objects-first in introductory computer science. *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, Nevada, United States 191-195.
- Dale, N. (n.d.a): *Course Content Survey Results (publisher's list group)*. <http://www.cs.utexas.edu/users/ndale/ContentResults2.html>. Accessed 4 April 2006.
- Dale, N. (n.d.b): *Course Content Survey Results (SIGCSE group)*. <http://www.cs.utexas.edu/users/ndale/ContentResults.html>. Accessed 4 August 2007.
- Dictionary.com (n.d.): *Dictionary.com Unabridged (v 1.0.1)*. <http://dictionary.reference.com/search?q=using>. Accessed 24 August 2006.
- DMLF (2006): *Danish Association of Datamatician Educators*. <http://www.dmlf.dk/>. Accessed 4 August 2007.
- Engel, G. and Roberts, E. (2001): *Computing Curricula 2001 Computer Science, Final Report*. [http://acm.org/education/curric\\_vols/cc2001.pdf](http://acm.org/education/curric_vols/cc2001.pdf). Accessed 28 August 2007.
- Haney, W., Russell, M., Gulek, C., and Fierros, E. (Jan-Feb, 1998). Drawing on education: Using student drawings to promote middle school improvement. *Schools in the Middle*, 7(3), 38–43.
- Hu, C. (2004): Rethinking of Teaching Objects-First. *Education and Information Technologies* 9(3): 209-218.
- ICALT'04 (2004): Proceedings IEEE international conference on advanced learning technologies. Kinshuk, C. Looi, E. Sutinen, et al. (eds)
- International handbook of universities (2005): Palgrave Macmillan, New York.
- ITiCSE'05 (2005): ITiCSE'05: Proceedings of the 10th annual conference on innovation and technology in computer science education.

- Koenigstein (n.d.): *Fachdidaktische Gespräche zur Informatik in Königstein*. <http://koenigstein.inf.tu-dresden.de>. Accessed 14 August 2007.
- Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003): The BlueJ system and its pedagogy, *Journal of Computer Science Education* 13(4): 249-268.
- Korhonen, A. and Malmi, L. (2004): *Kolin Kolistelut - Koli Calling 2004: Proceedings of the fourth Finnish/Baltic Sea Conference on Computer Science Education*, Report TKO-A42/04 edn, Helsinki University of Technology, Department of Computer Science and Technology, Helsinki, Finland.
- Lewis, J. (2000) Myths about object-orientation and its pedagogy", *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* , Austin, Texas, United States 245-249.
- Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., and Whalley, J. L. (2006). Research perspectives on the objects-early debate. In *Working Group Reports on ITiCSE on innovation and Technology in Computer Science Education* (Bologna, Italy, June 26 - 28, 2006)
- Milne, I. and Rowe, G. (2002): Difficulties in Learning and Teaching Programming – Views of Students and Tutors, *Education and Information Technologies*, 7(1):55-66.
- Moritz, S.H., Wei, F., Parvez, S.M. and Blank, G.D. (2005): From objects-first to design-first with multimedia and intelligent tutoring. *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* Lisbon, Portugal, 99-103.
- National centre for educational statistics (2007): <http://nces.ed.gov/index.asp>. Accessed 13 May 2007.
- National centre for education statistics (2007b): *COOL: College Opportunities Online Locator*. <http://nces.ed.gov/ipeds/cool/> Accessed 29 August 2007.
- Schulte, C. and Bennedsen, J. (2006): What do teachers teach in introductory programming? *Proceedings of the Second International Computing Education Research Workshop (ICER'06)*. Canterbury, United Kingdom, 17-28.
- SIGCSE mail-list (n.d.) Special Interest Group on Computer Science Education mailing list. <http://sigcse.org/join/list.shtml>. Accessed 29 August 2007
- SIGCSE'05 (2005): SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education, P. Tyman & D. Baldwin (eds).
- SIGCSE-members (2005): *Archives of sigcse-members@ACM.ORG* [Homepage of ACM Special Interest Group in Computer Science Education], [Online] <http://listserv.acm.org/archives/sigcse-members.html> Accessed 29 August 2006.
- Stemler, S. (2001): An overview of content analysis. *Practical Assessment, Research & Evaluation* 7(17). <http://pareonline.net/getvn.asp?v=7&n=17>. Accessed 29 August 2007
- Wallnau, L. B., & Gravetter, F. J. (2005). *Essentials of statistics for the behavioral sciences*. New York: Thomson Learning.
- Weber, R. P. (1990). *Basic Content Analysis*, 2<sup>nd</sup> ed. Newbury Park, CA.
- Yamane, T. (1973): *Statistics. An Introductory Analysis*, Harper & Row, New York.



# Contextualising information and communications technology in developing countries

**Meurig Beynon      Antony Harfield**

Department of Computer Science  
University of Warwick  
Coventry CV4 7AL, UK  
[wmb,ant@dcs.warwick.ac.uk](mailto:wmb,ant@dcs.warwick.ac.uk)

**Mikko Vesisenaho**

Department of Computer Science  
University of Joensuu  
PO Box 111, 80101 Joensuu, Finland  
[mvaho@cs.joensuu.fi](mailto:mvaho@cs.joensuu.fi)

## Abstract

This paper links two perspectives on the problems of introducing information and communications technology (ICT) and ICT education to developing countries. Ongoing projects aimed at establishing ICT provision for Tumaini University in Tanzania have led to the identification of a strategy ('the CATI model') that aspires at contextualising ICT in a progressive fashion, through activities that can be interpreted as importing, transferring and applying ICT. Independent research at Warwick has highlighted the way in which orthodox ICT-based education promotes a particular variety of learning, where knowledge that can be de-contextualised is privileged. The aspirations for CATI are reviewed with reference to an alternative conception of ICT rooted on a methodology for modelling with dependency ('Empirical Modelling'). An Empirical Modelling perspective on ICT is potentially seen as overcoming some of the obstacles to contextualising information and communications technology in developing countries. This potential is illustrated with reference to a model of the Linux vim editor that has been developed to bridge the gap between the cultures of the graphical user interface and the command line.

**Keywords:** Design, Experimentation, Human Factors, developing countries, ICT education, Contextualisation.

## 1 Introduction

This collaborative paper was motivated by a coincidental similarity between two diagrams. Whilst presenting their papers at the Technology for Education in Developing Countries (TEDC) conference in

Tanzania in July 2006, Vesisenaho and Beynon were struck by the resemblance between mental pictures they had developed independently to express distinct but potentially related issues concerning introducing ICT in the developing world.

Vesisenaho's diagram (Figure 1) represents 'the CATI model' – a strategy for managing the introduction of ICT, based on a classification that identifies four levels of technological adoption: contextualised, applied, transferred, imported (Vesisenaho, Lund and Sutinen 2006). The arrows to the left and right of this diagram respectively refer to planning and implementation phases, it being understood that the successful introduction of a technology relies on giving priority at the outset to contextualisation in every cycle of planning and implementation, and achieving this progressively through importing, transferring and application.

Beynon's diagram (Figure 2) was used to illustrate 'closed learning', one of two varieties of learning identified in Beynon (2006), and the kind of learning that is most characteristic of ICT-mediated education in the developed world. In this figure, the arrows to the left and right represent sequences of learning activities that are ranged within an experiential framework for learning ('the EFL'). The arrow on the left relates to activities that originate in the informal concrete subjective everyday world of experience and terminate in the abstract world of symbolic objective knowledge. The arrow on the right relates to activities that in effect unpack this symbolic objective knowledge by progressively exposing its connection with personal experience. Figure 2 can be interpreted as representing a particular model of education, whereby what has been learned and systematised is then taught through systematic explication.

Figures 1 and 2 both reflect their authors' common concerns: with the problems of promoting ICT in the developing world, and with the essential relevance of cultural context in addressing these problems. Both diagrams highlight activities that connect decontextualised knowledge and its associated

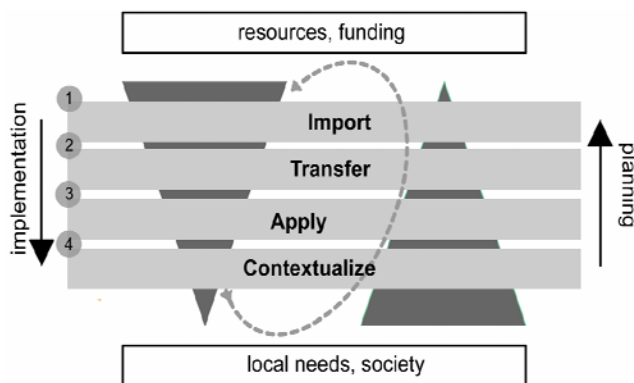


Figure 1. The CATI model.

technologies with a broader human-centred cultural context.

In Figure 1, the contextual aspects are represented by the box labelled 'local needs, society' at the bottom of the figure, and the decontextualised aspects by that labelled 'resources, funding'. Whilst the arrows to the right and left of the diagram might suggest a sequential process of planning followed by implementation, the dotted elliptical arrow stresses the iterative cyclic nature of the process. This is to acknowledge that context cannot be taken into account in advance of implementation, and that many contextualised uses of technology emerge only after speculative implementation. A key message of the CATI model is that the potential and prospects for contextualisation must be the primary consideration when developing new resources.

The CATI model highlights the significance of the activities by which a technology that has been developed in one culture is assimilated into a different culture. Figure 2 supplies a possible framework within which to conceptualise these activities. The progression of learning activities from the top to the bottom of the EFL broadly represent the way in which private interaction with an unfamiliar artefact can potentially evolve through broader interpretation and the acquisition of skill into interaction that engages other participants, possibly to take its place within a formal public context where interpretations acquire an objective abstract and symbolic character. As discussed in Beynon (2006), there is no reason in general why the trajectory of such learning should encompass symbolic representations. This is illustrated, for instance, by the enormous range and diversity of the cultural contexts in which music has developed, not all of which incorporate formal notations or precisely prescribed conventions for performance and whose interpretation necessarily has subjective and hermeneutic qualities. On this account, it is essential to consider forms of learning more general than closed learning in relation to the CATI model, which necessarily embraces varieties of contextualisation that do not conform to stereotypes for abstract communication based in logic and language.

Throughout this paper, our primary concern is for introducing products of the ICT culture of the developed world to the developing world. The predominating

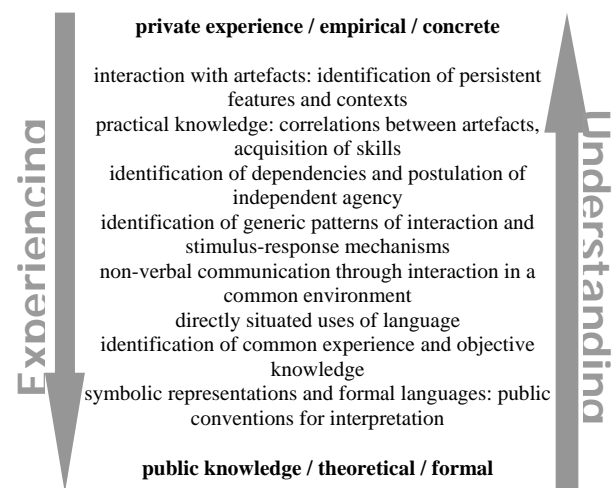


Figure 2. Closed learning in the EFL.

conception of ICT privileges formal and symbolic aspects of computer-related technology to such an extent that it is difficult to take full account of its human and experiential aspects, and hard to escape a closed learning paradigm. This leads us to argue that the way in which ICT is presently conceived presents serious obstacles to employing the CATI model, and that an approach to ICT that is rooted in a more pragmatic philosophical perspective is much better oriented towards the needs of CATI. This claim is developed with reference to an overview and appraisal of the CATI model (section 2), and a brief introduction to Empirical Modelling as a potential alternative foundation for ICT (section 3).

## 2 The CATI model

The CATI model can be seen as referring to any scenario in which a product developed in one cultural context is implanted in another. (As will be explained below, the term 'product' is to be very broadly interpreted.) *Import*, *transfer*, *application* and *contextualisation* represent four different forms that this implantation can take. To appreciate the distinction between them, it is helpful to consider the product in association with the entire body of physical and social adaptations by way of infrastructure and new practices that have contributed to its development and use. The extent to which different forms of implantation can be realised in practice depends critically upon the nature of the product and its relation to its context of development and use. A naïve visualisation of these forms is presented in Figure 3, where the ellipses on the left and right refer respectively to the existing and new potential contexts of use, and the 'cloud' icon depicts actual use in the new context.

**Import:** In this form of implantation, the product itself has been exported to a new context, but its essential infrastructure is so mismatched to its new setting that it cannot serve any of its original functions. Archetypal examples might be the export of a railway engine to a country without a rail network, or the acquisition of a computer merely as a status symbol.

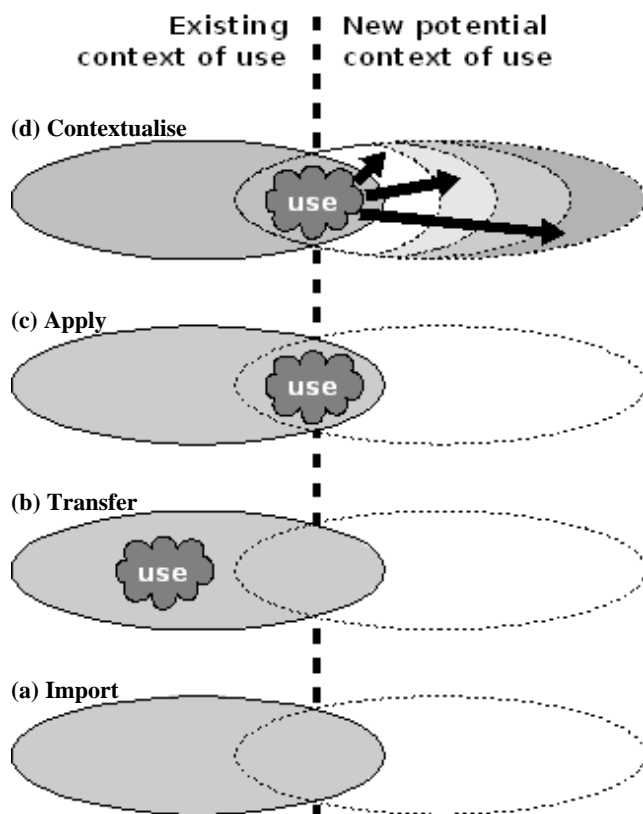


Figure 3. Varieties of product implantation within CATI.

**Transfer:** Here there is sufficient infrastructure in the new context to allow the product to be used in a fashion that might be appropriate in its original context but that does not serve a useful social or practical function in the new context. Such a form of implantation arises for example when a teacher who has been trained in a developed country tries to disseminate learning that is too specialised in character to be relevant to their native developing country.

**Apply:** In this scenario, the implanted product and its associated infrastructure are sufficiently well-matched to the new context for the product to serve some of its original functions in a useful fashion. For instance, certain basic IT functions served by a computer, such as word-processing, might be usefully applied even though its full potential as a programming device remains unexploited.

**Contextualise:** A product becomes contextualised if its original infrastructure can have a counterpart or even a complement within the new setting, whereby the product becomes integrated in its new context in such a way that new possibilities for its use and development can be envisaged. The adoption of suitable mobile phone technology in many parts of Africa, complemented by the use of solar recharging units, can at least partly exemplify successful contextualisation, though it has only a small element of local development.

The notion of contextualisation being invoked here involves aligning an implanted product and its infrastructure to its new context. This alignment ideally takes the form of meeting needs that are local and

locally defined, and in this way are sufficiently relevant to authentic personal goals to assure sustainability. This is to be distinguished from attempting brutal re-engineering of the new context to accommodate implantation of a product. By way of illustration, in discussing developments in Namibia, Matengu (2006) concludes that many technological tools used in Namibia are 'techno-economic imports', whose adoption was driven by motivations outside Africa and could be effected only by changing certain cultural and economic values, symbols and creations.

## 2.1 Appraising the CATI model

As the illustrative examples above show, the notion of implanting a product in a new context can be very broadly interpreted. The product itself may be a physical object, an organisational structure, or an abstract artefact. The context-switch may relate to transporting a product from one country to another, or from a standard classroom setting to a special-needs environment, or from one computing platform to another. The CATI model is conceived as more than a framework in which to interpret planning and implementing a physical ICT infrastructure in a developing country; it necessarily encompasses the implantation of products of many other kinds. This is just another way of interpreting contextualisation: the physical ICT infrastructure can only be sustained if useful software and meaningful computing practices can also be implanted in the new cultural setting.

The strategy – implicit in Figure 1 – of giving priority in planning to contextualisation was applied in teaching introductory computer programming at Tumaini University in Tanzania in 2004-5 (Sutinen and Vesisenaho 2006; Vesisenaho, Kempainen, Islas Sedano, Tedre, and Sutinen, 2006). This entailed abandoning the traditional 'theory-driven' approach to programming whereby students learn the operational semantics of basic syntactic constructs by building small generally reliable and predictable programs. The emphasis was instead placed upon a community-driven, needs-based approach to learning ICT consistent with the innovation systems approach for economic improvement advocated in the local region and in the wider Tanzanian context. This approach exploited concretising and visualisation tools together with field projects that gave the students a better grasp of concepts and semantics whilst making links with their previous knowledge, programming expertise and future working environment. The course succeeded in dispelling the notion of programming as 'becoming familiar with a list of abstract constructs', led to promising ideas and proposals for application, and stimulated some awareness of deeper contextual issues.

The difficulties encountered by Sutinen and Vesisenaho in applying the CATI model to ICT education are symptomatic of general problems associated with the communication of 'closed learning' discussed at length in Beynon (2006) (cf. Figure 2). The products of the developed world are typically the result of a deep and extensive process of adaptation to context that can be

traced as a progression of 'learning activities' that originate in concrete personal experience but terminate in sophisticated physical artefacts, working practices and abstract conceptions in the objective world. Such products are decontextualised in that they are conceived on a universal model for generic use and interpretation. At the same time, they are inseparably tied up with their implicit co-evolving context of infrastructure and protocol and on this account can be exceptionally difficult to contextualise in any other setting. Most significantly, it is the key imperatives of the developed world – such as the commercial drivers behind highly optimised design for mass markets – that promote such entangled interdependency between product and context.

The tethering of product to context is reinforced by the accepted conception of ICT. Modern computer science recognises no alternative to the fundamental notion of a computer program as a functional object characterised by certain well-specified patterns of use and interpretation. It would be considered bad practice to deploy a program without clearly specifying its context of use and interpretation. The result is a clear demarcation between the roles of the programmer and the end-user in engineering the context of use and interpretation. This presents major obstacles to several of the motivating objectives in introducing programming with contextualisation in the foreground. It makes it difficult to bring together the agendas of 'ICT for support' and 'ICT for education', and can be seen as accounting for what is identified in Tedre, Sutinen, Kähkönen and Kommers (2006) as "the clear failure of the computing community ... to translate computing methods and theory into terms that ordinary people can understand ... [and so] transfer the extraordinary benefits of its expertise to those who desperately need it".

## 2.2 Principles for pragmatic ICT?

The CATI model motivates a pragmatic approach to delivering ICT in developing countries in which concern for contextualisation takes precedence over programming theory. To date, the main focus in such projects has been on identifying applications with key relevance that can be addressed without requiring too deep a level of programming expertise (Sutinen and Vesisenaho 2006), and on making use of special technologies such as Lund's I-BLOCKS that provide a concrete setting in which the basic principles of abstract computer programming can be learnt (Nielsen and Lund 2006). More generally, a pragmatic perspective on ICT is closely associated with a constructionist outlook, whereby model-building with personally meaningful objectives is intended to promote the learning of computing skills.

Giving priority to contextualisation in approaching ICT has top-down and bottom-up aspects. The one involves conceiving and specifying applications that are appropriate in the local context and can serve as motivations for deploying ICT. The other is concerned with identifying concrete activities that can be linked with familiar aspects of the local culture whilst at the same time introducing concepts and skills connected

with ICT. If we accept the premise that computing is essentially about programs as functional objects, finding principles to support either approach is problematic. The history of the software crisis strongly suggests that neither approach is an effective way to make software development accessible to the novice or the end-user. The top-down and bottom-up approaches have their respective counterparts in development methods based on a waterfall model that aspire to create software from a functional specification, and in those based on eXtreme Programming principles that focus on creating functional objects from which applications can be synthesised.

As explained in Beynon (2006), contextualising products from the developed world in the developing world typically entails learning activities of an entirely different nature from those involved in disseminating products within the developed world. The latter is generally a form of closed learning, whereby what has been experienced in one context can be readily understood in another (cf. Figure 2). Starting from the local personal subjective concerns within the culture of a developing country demands an entirely different perspective on learning. While closed learning can be identified with what William James has characterised as understanding backwards, contextualising products in the developing world essentially involves what he describes as understanding forwards (James 1912, Beynon 2006).

Our accepted conception of ICT is fundamentally ill-suited to supporting understanding forwards. Programmed objects and tightly coupled functional specifications developed in association with a well-defined stable engineered context are the very embodiment of closed learning. An alternative framework for thinking about ICT is needed, in which it is possible to make sense of computer-based artefacts embracing multiple viewpoints, confusion and inconsistency such as is discussed in Sutinen and Vesisenaho (2006). This is not to deny that there has been practical success in deploying ICT to tackle relevant issues: witness educational applications based on modelling with spreadsheets; the development of microworlds; and computer-based tools that engage with concrete and experiential activities (such as music-making or creating digital pictures and games). Nonetheless, the fundamental science is an abstract theory of algorithms, of formal languages, and there is no accepted conceptual framework within which to privilege those aspects of computer use that are so significant outside the realm of closed learning.

## 3 An alternative ICT foundation?

Empirical Modelling (EM) is offered as an alternative framework for ICT that is well-suited to supporting a pragmatic perspective. It is founded upon primary concepts quite different from the 'objects' of an object-oriented paradigm: these are *observables*, *dependencies* and *agents*. Such concepts are adapted for dealing with more primitive aspects of a situation than the notion of object. With reference to the Jamesian philosophical

stance alluded to above, EM is first and foremost concerned with the what James identifies as the most primitive and fundamental ingredients of knowing: the conjunctive relations that connect two experiences and that are themselves given in experience. The idea that relationships can indeed be given empirically in this manner, without reference to some prior process of conceptualisation and rationalisation, is the key tenet of James's philosophy. Acknowledging this possibility can be seen as a prerequisite for recognising that knowledge need not be mediated through the rational world – as is essential in recognising the feasibility of 'understanding forwards' (cf. Figure 2).

An example of a conjunctive relation that defies rationalisation might be the association between 'flamingos and hedgehogs' and 'croquet mallets and balls' that is invoked in Lewis Carroll's *Alice in Wonderland* (1865). Any attempt to pin down such an association formally would be readily derailed by confusion and inconsistency, but the image nevertheless appeals vividly and directly to the reader's imagination. In order to use ICT to 'represent' such a relationship, the only effective recourse is to construct an artefact (such as a computer-generated animation or virtual reality) in which the connection between the two experiences is dramatised. This is the semantic principle that underlies EM. The meanings of a computer-based artefact are determined by the conjunctive relations it enters into in the experience of the human interpreter who interacts with it. This at once establishes the appropriate focus for understanding in relation to contextualisation: on the primary authority of individual personal experience and on the openness to negotiation of meaning that is required in assimilating the products of another culture.

The concepts of observable, dependency and agent are all implicated in building artefacts to establish conjunctive relationships. For this purpose, it is necessary to construct artefacts that are directly experienced to exhibit suitable patterns of state change on interaction. Something has to be seen or otherwise directly experienced (hence observables); there has to be scope for interactions that change observables in ways that are exploratory and potentially unpremeditated (hence agents); the artefact must respect the way in which changing one observable can directly impact on the value of another in a manner that can become familiar and predictable (hence dependencies). A familiar parallel for such activity is to be found in the construction and use of a spreadsheet. The fact that constructing spreadsheets is recognised to be one of the computing practices that has been most effectively exploited in domain learning (Baker and Sugden 2003) is further circumstantial evidence that EM has promise as an alternative foundation for ICT.

### 3.1 EM in relation to other approaches

In assessing the credentials of EM as a new basis for ICT, it is vital to understand its relationship to other pragmatic schools of thought. One significant feature of EM is that it is more than a way of thinking – it stems from practical origins, and has evolved in close

conjunction with model-building and project work in Computer Science at the University of Warwick over some twenty years (see the online EM archive cited in the references). Interpreting some practical uses of EM with reference to the CATI model may be helpful in understanding EM in relation to other approaches.

A parallel can be drawn between introducing ICT education to the developing world and restoring confidence to computer science students who have been alienated by conventional computer programming. Such students are often highly motivated by potential applications of ICT, but find traditional programming so uncongenial that they are reluctant to take up a technical ICT project. Several have later successfully pursued projects using EM principles and tools. In the process, they have typically traced stages similar to those represented in the CATI model.

A first step has been to identify where activities resembling those involved in their proposed application area are represented in an existing model. The nature of EM is such that they can experiment through interacting with an unfamiliar model with very limited understanding, so that they at first have only a mechanical idea of how to exercise it. In effect, the model may offer experiences by way of states and transitions that were meaningful to the original developer, but stimulate no conjunctive relations in the mind of the newcomer. At this point, the model may be seen as having been merely *imported*.

Before progressing to the next stage in model-building, it is necessary to become accustomed to the semantic framework within which EM is conducted. Students are typically locked into a mindset in which they have difficulty coping with the idea that the model is to be interpreted as offering a state-to-be-experienced rather than a functional object with specific closed behaviours and interpretations. In appreciating this semantic shift, it is helpful first to understand how the original developer of a model made sense of it – in effect, realising what conjunctive relation was in their mind in interacting with the model. Once the way in which such interpretation operated has been understood, if only partially, some element of *transfer* has taken place.

The step from transfer to application is often effected as much in the mind of the student as in practical development. A technical exercise in extracting the particular observables, dependencies and patterns of agency from the existing model may be involved but this is generally quite straightforward. Mentally, the significant step is recognising that the model admits new interpretations whose apprehension can be reinforced by making relatively superficial modifications, such as renaming observables, or making minor changes to their visualisation. In this fashion, the student comes to *apply* the model in their own context.

Once this process of partial model comprehension and appropriation has been successfully conducted, students typically gain rapidly in confidence. They will often repeat the process a few times with other models, each time enriching the semantics of their emerging artefact

within their own sphere of interest. It often comes as a surprise to the student that such an artefact can serve useful functions within their target application domain without having a fixed identity as a functional object or a precise specification for its use. Once this idea has been internalised, the student is ready to take the initiative in autonomous model-building, and those elements that were initially appropriated from other sources become absorbed, transformed and reinterpreted within the new context. This is indeed *contextualisation*.

#### 4 An illustrative example

The account of the relationship between the CATI model and EM in previous sections is not intended to be read as a philosophical essay – it also relates to a potential practical contribution. At this stage, there is no authentic practical evidence to support the application of EM principles and tools in the context of a developing country, but a concrete example that illustrates some of the motivating ideas may help to convince the sceptical reader that there is practical substance in our proposals. In fact, one of the difficulties in evaluating these proposals is that they presume a conceptual framework so radically different that its full implications for practice will take time to understand (Beynon 2007)

For the purposes of illustration, we shall consider cultural contexts that are topical in teaching programming to contemporary students of computer science. Many first year students of computing are quite unfamiliar with the notion of carrying out low-level programming through the command line rather than making use of a graphical user-interface (GUI) for high-level programming. In introducing Linux to such students, there is a cultural problem to which the thinking about CATI sketched above can be directly applied. It is essential to be able to convince students that the command line is not an obsolete notion, but a vital point of entry for fundamental programming tasks. For a generation accustomed to accessing resources through web applications, it is hard to envisage the context within which Linux itself was conceived and initially developed, where all commands and utilities had to be used through a textual interface. To cope with the demands of programming in full generality, the professional programmer still needs to be able to operate in such a context.

In this setting, an archetypal product to which the CATI model might be applied is a basic Linux command line editor, such as *vim*. Students familiar with GUI editors are typically reluctant to take a serious interest in *vim*. Whilst such students can be persuaded of the advantages of using basic Linux commands in file management, they naturally prefer to create files through direct manipulation of text. Referring to Figure 3(a) above, the status of *vim* is that of an *import* – a component of the Linux environment that is present but is perceived as having no useful application. The *transfer* of *vim* as a product is associated with a step that only some students take: finding out how the editor can be used to perform some routine editing task in a command line interface. In keeping with Figure 3(b), this represents a use of no

more than academic interest to the student. Even fewer students come to recognise that a line editor has practical advantages over a GUI editor in certain situations, and then come to *apply vim* as the editing tool of choice to perform a useful task (cf. Figure 3(c)). Exceptionally, some students recognise the potential for *vim* to serve much richer functions such as would be exploited in practice by a professional programmer or system administrator who frequently uses *vim* as the editor of choice. Such a use is exemplified in the work of a recent final-year project student at Warwick who, as an expert user, exploited the rich features of *vim* in developing a refactoring tool for the Python scripting language. Inasmuch as this project involved creating a GUI tool to support a highly technical underlying programming application, this can be seen as an example of *contextualisation* (cf. Figure 3(d)).

The examples of import, transfer, application and contextualisation of *vim* highlight a feature of traditional software that has been mentioned in section 3 above. Because of the way in which *vim* has been conceived and developed, it has an exceptionally well-defined functionality that suits it for application in a very specific context. It is significant that the expert use of *vim* cited above in connection with contextualisation is in the mainstream tradition of applications for a tool such as *vim*. It is associated with editing activity that supports programming at a level that is well-adapted to fundamental tasks, where there are typically strong and direct links between functions of the operating system and those of the program. The *vim* editor is a product that is deeply embedded in a particular infrastructure, and its contextualisation is difficult to imagine in isolation from this infrastructure. To fully appreciate the benefits of *vim*, the user needs to become familiar with the closed world that surrounds its conception and application. It is unclear to what extent it is necessary for an aspiring computing specialist brought up in a GUI culture to develop this familiarity, but there is no doubt that the conception of *vim* as a programming product is an obstacle to applying the principles behind the CATI model.

Empirical Modelling offers a practical alternative to treating *vim* as a program-like object. EM can be used to develop an artefact that captures the observables, dependencies and agency associated with using the *vim* editor. The resulting artefact, rather than being a program whose functionality and interpretation is tightly constrained by the cultural context in which it was developed, is a malleable source of experience for the user-modeller that affords a variety of different interpretations and potential applications. In the process of making such an artefact and developing scenarios for interaction and interpretation that exploit it, the cultures of the command line and the GUI are blended in a fashion that is not represented in the instances of import, transfer, application and contextualisation of *vim* cited above. The characteristics of the artefact itself, and the manner of its construction, are also much better suited to meeting the demands of the CATI strategy for contextualisation. The primary reason for this is that the

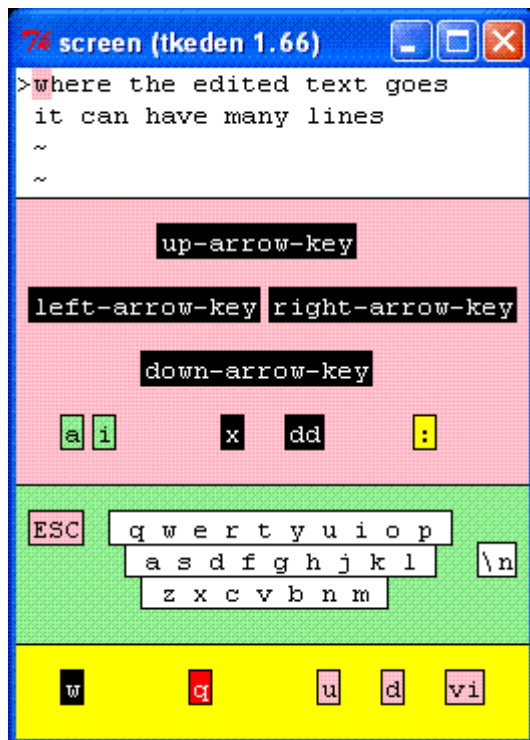


Figure 4. The vimodesBeynon2006 model.

vim artefact is being appraised from the perspective of a potential user within the target culture as a source of open experience rather than the embodiment of a closed functionality.

#### 4.1 Modelling vim as an EM artefact

Figure 4 is a screenshot of an artefact imitative of vim that has been constructed using EM principles and tools. The model – identified as vimodesBeynon2006 in the online EM archive – was developed incrementally over a few days in November 2006 in connection with a module that introduced Linux to first-year computer science students at the University of Warwick. The core of the model is a family of definitions that record the dependencies between key observables. These observables relate to many aspects of the state of the vim editor in use, where the concept of state reflects a holistic view of interaction with the application. Examples of observables include the content of the text that is being edited (as displayed in the top segment of the screen display in Figure 4); the physical form of the computer interface (as represented by the keyboard on the visual display); the modes of interaction with the editor, as registered in the mind of the user (each associated with one of the three panels in the screen display in Figure 4); and a basic repertoire of primitive editing operations (as represented by the buttons in these panels).

By way of illustration, the current line and column position are observables with direct counterparts `currline` and `colno` in the vim model. The current values of `currline` and `colno` determine both the character `currchar` that is currently under the cursor

and the location of the one-character sized window highlight on the display that shows the position of the cursor on the display:

```
%scout
window highlight = {
  type: TEXT
  frame: ([{colno.c,
            (currline - 1).r},1,1])
  string: currchar
  bgcolor: currentcolour
  fgcolor: "black"
  border: 0
};

%eden
currchar is substr(lines[currline],
                   colno,colno);
currentcolour is
(currentmode==reviewing) ? "pink" :
((currentmode==inserting) ?
 "lightgreen" : "yellow");
```

The background colour `currentcolour` of the highlighting window reflects the colour encoding of the panels associated with the various modes of the vim editor. Its current value depends upon the mode in which the editor is currently operating. Note also that, since the highlight window is opaque, its content must be that of the highlighted character `currchar`.

By capturing key observables and dependencies in an interactive environment, a family of definitions can mediate the interactions of many kinds of state-changing agent. In the early stages of development of the vim model, the relationships expressed in the sample definitions above can be validated by direct interaction on the part of the modeller to redefine observables such as `currline` and `colno`. In the model that is depicted in Figure 4, these redefinitions are effected indirectly by simulating the keyboard entry of characters by clicking the mouse on the display. This exemplifies a standard pattern of development in EM, whereby redefinitions that in the early stages are performed manually by the modeller are later mediated by ‘agents’ triggered via the interface, or indeed otherwise automated (as in the implementation of an undo procedure). The important feature of such a model is that observables and dependencies supply an environment for agent interaction that persists and is always open to direct access by the modeller. In a traditional program, in contrast, the actions are typically framed with complete prior knowledge of their intended functionality and with such emphasis on efficient accomplishment of purpose that the environment for interaction becomes no more authentic than the backdrop scenery in a theatre, whose role is but to create just such illusion as is needed to support the stage action.

The incremental development of the vim model in some respects resembles agile software development, but differs in its essential character. Agile software development generates prototype line editors, each with

well-specified but initially limited capabilities. By comparison, the *vim* model progressively embodies ever richer families of observables, more complex patterns of dependencies amongst observables, and more sophisticated kinds of agency. Different ways of configuring observables and dependencies and acting upon them serve to highlight different potential functionalities, but the development need not be driven by the explicit identification and implementation of specific objects and uses. Such an approach, where the initial and primary emphasis is upon the infinite possibilities for future interaction and interpretation, rather than the specific goals and constraints, is better suited to creative and exploratory development. It is also well-attuned to the priority given to contextualisation in the CATI strategy (cf. Figure 3(d)).

The *vimodesBeynon2006* model depicted in Figure 4 has no functional specification. Its behaviour and interpretation are products only of how the modeller decides to interact with it, so that it can be adapted for many different possible applications. For instance, there is a mode of interaction in which no redefinitions of key observables are made, and all changes of state are mediated through the model interface, and this corresponds to playing the role of a *vim* user (albeit one who has limited knowledge and privileges in respect of the full functionality of *vim*). Various extensions, adaptations and patterns of interaction might be appropriate in realising other objectives for the use of the model. As it stands, the model was primarily intended to familiarise the naive *vim* user with the different modes in which key presses can be interpreted. The model of the edited text is exceptionally simple, there being no more than four lines of a fixed length that are stored in a list [*line1*, *line2*, *line3*, *line4*]. The EM archive includes a variant of the model *vimodelBoyatt2007*, developed by Russell Boyatt, that comes closer to an authentic implementation of the editor. This model does not have such a limited model of the text. It is also based on a conceptualisation of the functionality that is aimed at the expert user rather than the novice. Consider, for instance, the way that – in *vimodesBeynon2006* – the button ‘*dd*’ within the interface in Figure 4 designates an atomic action to delete a line from the text. In a more appropriate mental model of the *vim* editor, entering the single character ‘*d*’ prepares for the deletion of entities other than lines alone, such as the word containing the current character subject to typing ‘*w*’.

The *vim* model admits many kinds of interaction other than the relatively orthodox uses discussed above. Boyatt, as a much more expert *vim* user than Beynon, was able to identify limitations in Beynon’s understanding from the *vimodesBeynon2006* model. For instance: the conception of the modes in *vim* is somewhat naive; the failure to appreciate that inputs such as ‘*dw*’ are possible has significant ramifications; and the behaviour of the cursor on completion of an edit has not been faithfully observed. The first two of these issues exemplify problems that arise where communication between a sophisticated and less

developed culture is concerned. Beynon’s model is based on a workable conception of *vim* that is not dissimilar to that outlined for the benefit of the Linux novice in Joy, Jarvis and Luck (2002), for instance. In this respect, it is not so much *wrong* as limited, and has problematic aspects when more advanced use of the editor is taken into account.

The third issue noted by Boyatt is of course a genuine error that renders Beynon’s *vim* model incorrect. Because of the character of the model, this error is easily remedied by adding a single redefinition to the action of the agent that switches from the editing to reviewing mode. The use of dependency to model the environment for agent interaction plays a crucial role here in ensuring that such errors can be remedied in a way that has retrospective effect. This scope for discovering and fixing conceptual errors ‘on-the-fly’ is the practical expression of the notion of ‘understanding forwards’. This is in contrast to traditional programming, where it is typically not easy to address the consequences of conceptual errors – because they relate to actions and relationships outside the scope of those explicitly considered in the specification, and are therefore not the subject of special attention in the construction of a program. Insisting that conceptualisation must precede insightful action is characteristic of ‘understanding backwards’.

## 4.2 Supporting contextualisation

The openness of the *vimodesBeynon2006* model to many interpretations and modes of interaction stems from its status as an EM model as distinct from a program. The above discussion reveals the potential of the *vim* model to serve as a boundary object (Star 1989) between expert and novice users. It can also be viewed as a boundary object for the GUI and command line cultures, integrating elements of command line syntax that are explicit in the interface through which the definitions, functions and actions in the model are specified and revised, with the buttons and other widgets on the screen providing visual counterparts for these actions that can be carried out by direct manipulation. Note that the kind of redefinition that is being considered in interacting with and interpreting the model is much richer than can be encompassed by the use of a *vim* program – or even by the *vilearn* extension of the *vim* editor designed for tutorial use. For instance, the entire sequence of definitions used to construct the model, as recorded in the EM archive, traces the *vimodesBeynon2006* model from its embryonic state as a mere specification of four lines of text. Studying the interaction and interpretation embodied in this trace serves to introduce EM principles and tools (with particular emphasis on the use of the definition-based notation Scout for screen layout).

To further dramatise the observation that the model has no functional specification, we consider one way in which the model can be readily adapted to serve an entirely different function from any of those mentioned above.

Figure 5 shows a variant of `vimodesBeynon2006` that is derived from the original model by redefining a few observables and making a few new definitions:

```

line4 = "aaaaaaaaaaaaaaaaaaaaa";
line1 is strtobin(line4);

factors_M is factors(M);
isprime_M is factors_M#==1;
M is binstrtoint(line1);

ansstr is (isprime_M) ? "" : "n't" ;

line2 is "The number " // str(M)
        // " is" // ansstr // " prime";
line3 is "It has " // str(line1#)
        // " binary digits";

func factors {
    para n;
    auto i, result;
    for (i=1; i<=sqrt(n); i++)
        if (n==(n/i)*i)
            result = result // [i];
    return result;
}

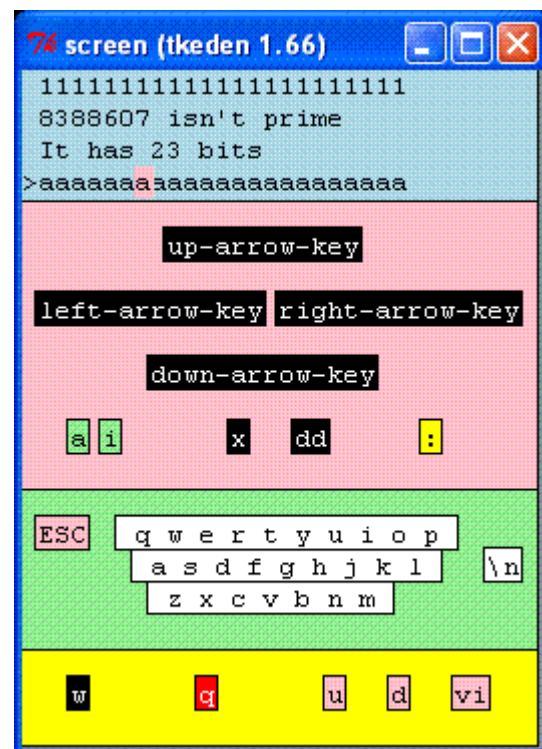
```

In this variant the text string at `line4` is translated to a binary string by converting each character to 0 or 1 according to the parity of its ASCII code. This string is recorded in `line1`. Whether or not the binary number encoded at `line1` is prime is then registered in the text in `line2` and `line3`. Using the `vim` interface to edit `line4` is an indirect way of manipulating the binary number at `line1`. The definitions needed to effect this transformation are as listed above. They make use of the user-defined functions `stringtobin`, `factors` and `binstrtoint` that are encoded procedurally in a style that is here illustrated in the specification of `factors`. Though this is no more than a playful example of how readily transformations of the original model can be developed, it also serves to highlight how EM enables the flexible adaptation to new kinds of interaction and interpretation such as are directly relevant to the technical challenges to be confronted in contextualising following the CATI strategy (cf. Figure 3(d)).

## 5 Concluding remarks

Further tool development and rigorous empirical research are needed before the merit and novelty of EM as a practical basis for ICT education can be decisively established. By way of conclusion, it is helpful to review some of the key concerns that are relevant where contextualisation of ICT is concerned.

As discussed above, in understanding the many forms that contextualisation of ICT products can take it is essential to go beyond the perspective that views the ICT culture as primarily rooted in ‘computational thinking’ (Wing 2006). In trying to develop new programming languages that are more accessible to the



**Figure 5. A variant of the vimodesBeynon2006 model.**

end-user, it has proved exceptionally difficult to moderate the attention that must be paid to the abstract computational agenda. Where languages like Logo initially focussed on simple procedural constructs, their descendants have typically imported a far more complex infrastructure of abstractions and operations that may in some respects hide low-level complexity, but pose new conceptual problems, such as are associated with objects, methods, inheritance and the like. Whilst the methodology for modelling with dependency in EM (Beynon 2007) helps to shift the emphasis from abstract computational mechanisms to the meaningful semantic relations, it is still necessary to frame the formulae or in some way program the functions that define dependencies. It remains to be seen to what extent this ingredient can be represented in ways that avoid computational artifice. The relative success of spreadsheets in educational applications offers some encouragement (Baker and Sugden 2003). Perhaps the most promising feature of EM tool development to date is that new expressive possibilities can be addressed by introducing a new notation that affords new metaphors but is conceptually like any other notation for formulating dependencies.

A significant element in contextualisation is being able to use an artefact as a boundary object (Star 1989) that can serve in different roles. The way in which an EM artefact can be fashioned using essentially the same principles by design participants acting in different roles has been identified elsewhere (Beynon and Chan 2006) as a good basis for model-building in the constructionist mould. According to the constructionist ideal, the learner as model-builder has in effect to play the roles of student/user, developer/modeller, and teacher/designer.

As discussed in Beynon and Harfield (2005), the same principles that enable one student to contextualise ingredients of a model that were first assembled by another can also bring coherence to the environment in which these participants interact.

The breadth of the demands placed upon ICT in CATI is best understood by recognising that contextualisation demands that ICT artefacts are viewed as ‘constructions’ in a broad sense. As pointed out by Latour (2003), the notion of constructivism that is being invoked here has proved to be most vexatious. In discussing this issue, Latour speaks with caution about the prospects of “[lifting] the curse on the theory of action implied by the many metaphors of construction”. Beynon and Harfield (2007) position EM in relation to other conceptions of constructivism by assessing the extent to which it meets the criterion set out in Latour (2003): that of strengthening five guarantees he identifies *taken together*. The relevance of these guarantees to the theme of this paper is that they bear directly on the construction of a “good common world”.

## 6 Acknowledgements

We are indebted to Russell Boyatt for his contribution to the study of vim models in section 4, and to several anonymous referees for most helpful feedback on earlier drafts of the paper.

## 7 References

- Baker, J.E., Sugden, S.J. (2003) Spreadsheets in Education – the First 25 Years. *Spreadsheets in Education* 1(1), 18-43.
- Beynon, W.M. (2007) Computing technology for learning – in need of a radical new conception. In *Journal of Educational Technology & Society*, 10 (1), 94-106.
- Beynon, W.M. & Harfield, A. (2007) Lifelong Learning, Empirical Modelling and the Promises of Constructivism. *Journal of Computers*, 2(3), 43-55.
- Beynon, W.M. (2006) Towards Technology for Learning in a Developing World. In *Proceedings of TEDC'06*, Iringa, Tanzania, 88-92.
- Beynon, W.M. & Chan, Z.E. (2006) A conception of computing technology better suited to distributed participatory design. NordiCHI Workshop on Distributed Participatory Design, Oslo, Norway, October 2006.
- Beynon, W.M., & Harfield, A. (2005) Empirical Modelling in support of constructionism: a case study. In *Proceedings of ICALT'05*, Kaohsiung, Taiwan, 396-398.
- Carroll, L. (1865) *Alice in Wonderland*. Mad Hatter Publications, London.
- James, W. (1912) *Essays in Radical Empiricism*. Longmans, Green, and Co., London.
- Joy, M., Jarvis, S. & Luck, M. (2002). *Introducing UNIX and Linux*. Basingstoke: Palgrave Macmillan
- Latour, B. (2003) The promises of constructivism. In Evan Selinger, Don Ihde (eds.) *Chasing Technoscience: Matrix for Materiality*. Indiana University Press, 27-46.
- Matengu, K.K. (2006). *Adoption of ICT at schools in core and periphery settings of Namibia: Exploring innovation, technology policy and development issues*. PhD Thesis. Aachen: Shaker Verlag.
- Nielsen, J. & Lund, H.H. (2006) Contextualised design of African I-BLOCKS. In *Proceedings of TEDC'06*, Iringa, Tanzania, 14-18.
- Star, S.L. (1989) The structure of ill-structured solutions: boundary objects and heterogeneous distributed problem solving. In L. Gasser and M. Huhns (eds.) *Distributed artificial intelligence*, Vol. II, London, Pitman, 46.
- Sutinen, E & Vesisenaho, M. (2006). Ethnocomputing in Tanzania: Design and Analysis of a Contextualised ICT Course. *Research and Practice in Technology Enhanced Learning* 1(3), 239-267.
- Tedre, M., Sutinen, E., Kähkönen, E., & Kommers, P. (2006) Ethnocomputing: ICT in Social and Cultural Context. *Communications of the ACM* 49(1), 126-130.
- Vesisenaho, M., Kemppainen, J., Islas Sedano, C., Tedre, M. & Sutinen, E. (2006) How to Contextualise ICT in Higher Education: a Case Study in Tanzania. *African Journal of Information & Communication Technology* (AJICT), 2(2), 88-109.
- Vesisenaho, M., Lund, H.H. & Sutinen, E. (2006) Contextual analysis of Students' Learning during an Introductory ICT course in Tanzania. In *Proceedings of TEDC'06*, Iringa, Tanzania, 9-14.
- The vilearn tutorial at the URL: <http://www.freebsdsoftware.org/editors/vilearn.html> (retrieved December 5, 2007)
- Wing, J.M. (2006) Computational Thinking. *Communications of the ACM*, Vol 49(3), 33-35.
- EM website and model archive at the URL: <http://www.dcs.warwick.ac.uk/modelling> (retrieved October 31, 2007)

# Incorporating Programming Strategies Explicitly into Curricula

**Michael de Raadt, Mark Toleman**

School of Information Systems  
Faculty of Business  
University of Southern Queensland  
Toowoomba, Queensland, 4350, Australia  
`{deraadt,markt}@usq.edu.au`

**Richard Watson**

Department of Mathematics and Computing  
Faculty of Sciences  
University of Southern Queensland  
Toowoomba, Queensland, 4350, Australia  
`rwatson@usq.edu.au`

## Abstract

An experiment was conducted to test a curriculum that explicitly incorporated programming strategies in lectures, written course materials, exercises and assessment. A control curriculum was also established to allow for comparison and isolation of effects. The two curricula were delivered to two groups of volunteer students who had no previous programming experience. The experimental group showed understanding and application of programming strategies, used the vocabulary plans in interviews and showed greater confidence in their solutions to problems. This suggested that explicit incorporation of programming strategies into an introductory programming curriculum has the potential to improve outcomes for novice programmers.

**Keywords:** Introductory programming, curriculum, programming strategies.

## 1 Introduction

An important dimension identified in literature by Robins, Rountree, & Rountree (2003) is the knowledge-strategy dimension. *Knowledge* involves the declarative nature of a programming language while *strategies* describe how programming knowledge is applied (Davies, 1993). Programming strategies are made up of plans (Soloway, 1985) (or schema or patterns) and the associated means of incorporating these into a single solution. Soloway (1986) suggests:

...language constructs do not pose major stumbling blocks for novices... rather, the real problems novices have lie in “putting the pieces together,” composing and coordinating components of a program. (p. 850)

Soloway then suggests teaching should reach beyond a focus on syntax (as programming knowledge) and focus on programming strategies. Recent studies (Lister et al., 2004; Whalley et al., 2006) have suggested novice programming knowledge can be fragile, so it is important to focus on both programming knowledge and strategy in

curricula. See de Raadt (2007b) for an overview of recent experiments in this area.

de Raadt, Tolman and Watson (2006) place problems faced by programmers on a scale as follows.

- **System Level Problems**

Problems at this level are large in scale and usually unique. An example of a problem at this level might be designing an accounting system for a large corporation. Students generally study problem solving at this level in a systems analysis course.

- **Algorithmic Level Problems**

Problems at this level are identifiable parts of a greater problem. (In an academic setting they may be addressed independently.) For such problems a solution is usually achieved by adopting well refined algorithms, widely used in the programming community. A novice may be able to start using such strategies at the end of an initial course in programming and may use them in greater depth in a second or third course in programming.

- **Sub-algorithmic Level Problems**

Problems at this level are at their most basic. Attempting to decompose and describe a problem below this scale will lead to syntactical definitions. Examples of problems at this scale are avoiding division-by-zero, achieving repetition until a sentinel is found, and so on. This level of problem solving is particularly relevant to novices in their initial exposure to programming. This level is perhaps the least recognised yet most fundamental to good programming problem solving.

Another important dimension relevant to this experiment defines how instruction is delivered, which can be described as being implicit, explicit, or a combination of these. *Explicit* instruction involves the instructor openly describing, usually in some documented form, what the student is to learn and how to go about that learning. *Implicit* instruction creates a scenario where a student is expected to undertake new learning, or extend previous learning, without being given a full context for what they are to learn or how. From the results of an experiment conducted by Biederman and Shiffrar (1987), Baddeley (1997) suggested a short period of explicit instruction can be more effective than months of implicit learning. Experiments by Reber (1993) showed students can learn

through implicit-only means, but this leads to a poor understanding of the underlying systems involved. Traditional curricula tend to rely on novices acquiring programming strategies implicitly.

A previous study (de Raadt, Toleman, & Watson, 2004) investigated an introductory programming course where novices were expected to learn programming strategies implicitly. Novices who participated in the study were asked to create a solution to a simple averaging problem. Solutions were scrutinised under Goal/Plan Analysis (Soloway, 1986) to measure application of strategies. Only one of 42 novices demonstrated application of all expected strategies. Students' solutions showed flaws in initialising variables, using a correct repetition strategy, guarding against events such as division by zero, and merging strategies that should be achieved together. These flaws implied weaknesses in the curriculum being delivered to the students at the time.

A second study (de Raadt, Toleman, & Watson, 2006) uncovered a model of expert programming strategies at a sub-algorithmic level based on plans described by Soloway (1986). These strategies can be explicitly expressed. This study suggested that the explicit inclusion of programming strategies should be attempted as it may:

- improve outcomes for students,
- establish a vocabulary for programming strategy dissemination, and
- allow students' programming strategy skills to be assessed.

This current experiment was conducted to discover if programming strategy instruction can be explicitly incorporated into an introductory programming curriculum, and if this is possible, what effects can be observed. Two curricula were designed to allow comparison and isolation of effects. One curriculum included explicit programming strategies while the second relied on implicit learning of programming strategies. Each of these curricula was delivered over a single weekend and followed by a series of one-on-one interviews with participants. No credit was awarded to participants; participants gained the experience of learning programming.

## 1.1 Research Questions

This experiment was motivated by the following interrelated questions (answered in section 5).

- Can programming strategies be explicitly incorporated into an introductory programming curriculum?
- What is the significance of the time consumed by this additional instruction?
- Can programming strategies explicitly taught in an introductory programming course be assessed?
- What impact does explicit strategy instruction have on students and their problem solving ability when compared to an implicit-only approach?

- Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?

In section 2, details of the experimental curriculum are described. Section 3 describes how the experiment was undertaken. Results of the experiment are displayed and discussed in section 4. Section 5 answers the research questions and concludes with implications and future work.

## 2 Description of Curricula

A base curriculum was created that contained programming strategy instruction explicitly. This curriculum is described further in this section and is included in full in a working paper (de Raadt, 2007a). From this, a second curriculum was created by identifying and removing programming strategy instruction components.

### 2.1 Incorporating Explicit Programming Strategies

In this experiment Soloway's *plans* were chosen over *patterns*, even though patterns have become more widespread in recent years. Patterns are bound to the Object paradigm and require a pattern language for application. Plans can be used in multiple paradigms, including the Object paradigm. Plans can be expressed simply, particularly at a sub-algorithmic level. In saying this, the focus of this research is not on the type of strategies that are taught but on *how* they are taught, and outcomes for students from that. It is likely that patterns could be used to achieve the same programming strategy understanding for students as plans. From this point on the term *plan* is used to represent a specific form of strategy and the term *strategy* is used in its more generic sense.

Programming strategies are explicitly incorporated into the curriculum in a number of ways. These are described in subsections 2.1.1 to 2.1.3.

#### 2.1.1 Identifying Strategies in the Curriculum

A book of written study materials was created and hardcopies were given to participants. Lecture slides were created based on the content of the written study materials. The lecture slides were used during lectures. In these written materials and lectures the strategies incorporated in the curriculum were named, their benefits were explained, and examples of their application were shown. Figure 2.1 shows a section of the written materials provided to students. In this example the Guarded Division Plan is identified. An explanation is given for why this plan is used, including a reference to an earlier mention of the consequences of dividing by zero. The description tells how the strategy is implemented and an example coded implementation of this strategy is shown. As well as introducing strategies, the descriptions also covered the means of integrating these strategies through abutment, merging and nesting (Soloway, 1986, p. 856).

## 10.5 Guarding Division

One application of an `if` statement is to prevent code which could result in unpredictable behaviour or cause the program to crash while being executed. Previously we saw how dividing by zero can produce an unusable result. In some programming languages the effects can be even more severe. It is recommended that you always test the divisor (the second, right-hand operand) before a division operation takes place. If the divisor is zero, division should be avoided.

```

01 <html>
02   <head>
03     <script type="text/javascript">
04       var number = 0;
05
06       number = parseInt(prompt("Enter a number for division"));
07       if(number != 0) {
08         alert(100 / number);
09       }
10       else {
11         alert("Dividing by zero causes problems");
12       }
13     </script>
14   </head>
15   <body>
16     Guarding division example
17   </body>
18 </html>

```

Code Example 10.5: The numerator of a division should always be tested before the division

### Exercise 10.5

Using your template, create a program that will prompt the user to enter a pre-calculated *sum* of numbers and pre-calculated *count* of numbers. Calculate the *average* (the sum divided by the count). How should your program behave if the user enters zero for the count of numbers?

Figure 2.1. An extract from the written course materials showing explicit incorporation of a problem solving strategy instruction

### 2.1.2 Paper Exercises and Practical Computing Tasks

At the end of each module students were asked to complete paper exercises and computer based tasks that reinforced the content delivered in lectures and allowed students to experience the practical implementation of the strategies covered. Instructions for these exercises and tasks were set out in the written materials, such as Exercise 10.5 shown in Figure 2.1. The exercise shown prompts users to explore Guarding Division. In other exercises students are prompted to experiment with the outcome achieved when the strategy is not applied or poorly applied. During the course, as with any normal introductory programming class, the instructor was on hand to answer questions and guide students. In most cases the exercises and tasks given were common to both curricula. In the curriculum without explicit programming strategies students were expected to learn the required programming strategies implicitly.

### 2.1.3 Assessment of Programming Strategies

At the end of the course, students were asked to complete the same three programming tasks that were given to experts in the previous study with experts (de Raadt, Toleman, & Watson, 2006). These tasks were used as a formal assessment at the end of the course under exam conditions. As well as testing participants' abilities, this was done to explore the potential to assess programming

strategies as part of a course. The strategies necessary to solve the final assessment problems had been shown as examples and in exercises and programming tasks.

## 2.2 Format of the Curriculum

The curriculum is based on a traditional curriculum that reveals parts of a given language in a sequence, with new knowledge of language concepts being dependent on previously covered knowledge. In this format, explicitly incorporating programming strategies depends upon certain underlying knowledge being taught beforehand. For instance, for the Guarded Division plan to be introduced, knowledge of variables, operators and selection must be covered first. Looking at the titles of the modules of the course shown in Table 2.1 gives little clue that explicit programming strategies are involved.

Basing the experimental curriculum on a traditional curriculum allowed the creation of a second curriculum without explicit programming strategies. In a non-experimental setting, the format of the curriculum could change. For instance, the structure of the course could be governed by the strategies themselves instead of the underlying language; in this case strategies could be introduced then underlying language knowledge could be taught. If an objects-first approach is taken, strategies could be used at other stages.

### 2.3 Philosophy behind the Experimental Curriculum

The curriculum was designed to be short and to allow students to reach programming strategies as soon as possible. The curriculum would not be effective in teaching longer courses, although the ideas used in the explicit incorporation of programming strategies could be applied to longer curricula.

The curriculum focused on programming strategies, with only a minimal covering of the knowledge components on which the covered strategies are dependent. Knowledge content was included if it was fundamentally important for learning the later programming strategies. Later exercises focused on the application of programming strategies. For those who had not been explicitly instructed in programming strategies, this was their opportunity to implicitly learn the needed strategies

**Table 2.1. Comparison of the two curricula tested (items with ~~strike through~~ were absent in control curriculum)**

Module	Section	Curriculum A (with Explicit PSS)	Curriculum B (without Explicit PSS)
<b>1</b>		<b>First JavaScript Program</b>	<b>First JavaScript Program</b>
	1.1.	Hello World!	Hello World!
	1.2.	JavaScript and HTML	JavaScript and HTML
	1.3.	Statements	Statements
<b>2</b>		<b>Calling Functions</b>	<b>Calling Functions</b>
	2.1.	alert()	alert()
<b>3</b>		<b>Values</b>	<b>Values</b>
	3.1.	Numbers	Numbers
	3.2.	Strings	Strings
	3.3.	Booleans	Booleans
<b>4</b>		<b>Variables</b>	<b>Variables</b>
	4.1.	What are Variables	What are Variables
	4.2.	Identifier Rules	Identifier Rules
	4.3.	Declaring Variables with var	Declaring Variables with var
	4.4.	Undefined	Undefined
<b>5</b>		<b>Assigning Values</b>	<b>Assigning Values</b>
	5.1.	Dynamic Typing	Dynamic Typing
	5.2.	typeof	typeof
	5.3.	Initialising Variables	<del>Initialising Variables</del>
<b>6</b>		<b>Operations</b>	<b>Operations</b>
	6.1.	Arithmetic Operators	Arithmetic Operators
	6.2.	Division by Zero – infinity	<del>Division by Zero – infinity</del>
	6.3.	Postfix Operators	Postfix Operators
	6.4.	Relational Operators (incl. Equality)	Relational Operators (incl. Equality)
	6.5.	Logical Operators	Logical Operators
	6.6.	String Operators	String Operators
<b>7</b>		<b>Abutment</b>	<b>Abutment</b>
<b>8</b>		<b>Debugging</b>	<b>Debugging</b>
			<del>Exercise 8.3</del>
<b>9</b>		<b>Functions that Return Values</b>	<b>Functions that Return Values</b>
	9.1.	prompt()	prompt()
	9.2.	parseInt() and parseFloat()	parseInt() and parseFloat()
<b>10</b>		<b>Selection</b>	<b>Selection</b>
	10.1.	The if Statement	The if Statement
	10.2.	The if-else Statement	The if-else Statement
	10.3.	Indenting and Formatting	Indenting and Formatting
	10.4.	“Dangling else”	“Dangling else”
	10.5.	Guarding Division	<del>Guarding Division</del>
<b>11</b>		<b>Repetition (Loops)</b>	<b>Repetition (Loops)</b>
	11.1.	while Loop	while Loop
	11.2.	Sentinel Controlled Loops	<del>Sentinel Controlled Loops</del>
	11.3.	for Loop	for Loop
	11.4.	Counter Controlled Loops	<del>Counter Controlled Loops</del>
	11.5.	Finding the Maximum/Minimum	<del>Finding the Maximum/Minimum</del>
	11.6.	Nesting and Merging	<del>Nesting and Merging</del>
<b>12</b>		<b>Arrays</b>	<b>Arrays</b>
	12.1.	Declaring Arrays	Declaring Arrays
	12.2.	Accessing Array Elements	Accessing Array Elements
	12.3.	Initialising Arrays	Initialising Arrays
	12.4.	Arrays for Values	Arrays for Values
	12.5.	Arrays for Categories	Arrays for Categories
	12.6.	Counting Values in a Set	<del>Counting Values in a Set</del>

through practical exercises. The assessment at the end of both forms of the course focused on the analysis of programming strategy skills developed through the course. In a non-experimental course the focus of exercises and the weighting of examination questions would be more balanced between knowledge components and programming strategies.

## 2.4 Language Used with Experimental Curriculum

JavaScript was used as the language to support the instruction of the curriculum. In their essential form, programming strategies are language independent and examples could be given in almost any language. Soloway and his colleagues used Pascal and Lisp to illustrate programming strategies. The authors have used C/C++ to exemplify programming strategies in other work.

JavaScript was chosen for this experiment for the following reasons:

- potential to reach important concepts rapidly;
- simpler to practice than a compiled language;
- attractive to volunteers;
- allows expression of programming strategies with a programming language not previously used for this purpose.

## 3 Methodology

The method of experimentation began with preliminary demographic, experience and confidence measurements. An examination of programming strategies was conducted at the end of each weekend. In the weeks that followed the two weekend sessions participants were invited to an interview in which they were asked questions about their solutions to gauge their understanding of the strategies that were being tested.

### 3.1 Volunteer Participants

Participants were volunteers from the student body at the University of Southern Queensland, and were recruited by posters hung around the university campus and by emails sent to former students of two computing concepts courses for non-computing students.

Participants were asked to undertake an initial survey that gathered demographic data, computing experience, past programming experience and a measure of computing confidence.

This initial data was used to filter students who had previous programming experience. Students with *no* previous programming experience were sought in order to set a baseline for all participants. Volunteers with previous programming experience were asked to withdraw.

A number of the volunteers withdrew from the weekend courses, mostly due to personal reasons, giving notice before the start of the experiment. A number of other volunteers failed to attend the course, which was unexpected, and reduced the group of volunteers to eight in two groups of four, divided on a self-selecting basis.

One of the participants who attended the first weekend had completed a previous course in computer programming and arrived after being asked by email not to attend. Results were collected from this participant but are not aggregated with other participants in this experiment.

### 3.2 Setting

The two weekend courses were conducted in a computing lab. This room included facilities for lecturing, computers for students to undertake practical exercises, and desk space between computers for students to complete paper-based exercises.

The two curricula were delivered on consecutive weekends. The curriculum without explicit programming content was delivered first and this was followed the next weekend by the curriculum with explicit programming strategies. The ordering of the two curricula was arbitrary.

The two days of each weekend were divided into sessions; with each session covering one to four modules of the course (see the schedule in section 3.4). Each session consisted of an initial lecture with questions encouraged from students. This was followed by paper tasks and practical programming tasks. Later in the course, tasks that involved programming strategies were used. Students were given breaks between sessions.

### 3.3 Demographic, Experience and Confidence Measures

A number of demographic, experience and confidence measures were conducted via a web survey presented to students when they volunteered. Participants were asked questions about:

- gender;
- age;
- computing experience;
- previous programming experience; and
- computing confidence.

Details of specific questions are given in de Raadt (2007a). Computing confidence was captured using a test created by Cretchley (2006), which has proven to be a reliable predictor of computing confidence in the past.

### 3.4 Schedule of Course Delivery

The schedule for both weekends was identical except where programming strategy content was covered. In Table 3.1, content covering programming strategies is highlighted and was covered only in the course with explicit instruction of programming strategies. Participants undertaking the course without explicit programming strategy content were intended to be attempting practical exercises during these times. One of the aims of the experiment was to determine if this additional content would impact on the balance of time allowed for lecture instruction versus exercises and practice. For this reason the schedule was followed as closely as possible on both weekends.

**Table 3.1. Schedule for Weekend Courses**

Session	Saturday Content
10:00 – 11:15	Introductions 1 First JS Program 1.1 Hello World 1.2 JavaScript and HTML 2 Calling Functions 2.1 alert()
	3 Values 3.1 Numbers 3.2 Strings 3.3 Booleans 3.4 Undefined 4 Variables 4.1 What are Variables 4.2 Identifier Rules 4.3 Creating variables with var 5 Assigning Values 5.1 Dynamic typing 5.2 typeof 5.3 Initialising Variables
11:30 – 13:00	6 Operations 6.1 Arithmetic Operators 6.2 Division by Zero - Infinity 6.3 Postfix Operators 6.4 Relational Operators (incl. Equality) 6.5 Logical Operators 6.6 String Operators 7 Abutment 8 Debugging 9 Functions that Return Values 9.1 prompt() 9.2 parseInt()
	10 Selection 10.1 The if Statement 10.2 The if-else Statement 10.3 Indenting and Formatting 10.4 "Dangling else" 10.5 Guarding Division
13:30 – 14:45	<b>Sunday Content</b>
	11 Loops 11.1 while Loop 11.2 Sentinel Controlled Loops 11.3 for Loop 11.4 Counter Controlled Loops 11.5 Finding the Maximum 11.6 Nesting and Merging 12 Arrays 12.1 Arrays for Values 12.2 Arrays for Categories 12.3 Counting Values in a Set
13:30 – 14:45	Testing

### 3.5 Administering the Final Assessment

After lunch on the Sunday of each weekend course, participants were asked to complete the three programming tasks previously given to experts (de Raadt, Toleman, & Watson, 2006). Each problem was presented on a single sheet of paper with lines below to complete the solutions to the problems (solution sheets are shown in de Raadt (2007a)). Participants were able to use as much time as was needed to complete problems.

#### Problem 1

*Read in 10 positive integers from a user. Assume the user*

*will enter valid positive integers only. Determine the maximum.*

#### Problem 2

*Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.*

#### Problem 3

*Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.*

The solutions produced were examined using Goal/Plan Analysis to test for the presence or absence of expected plans. This was conducted in the same manner as the earlier experiment with experts. The expected strategies and means of integration are given with results.

### 3.6 Post-Experiment Interviews with Participants

In the 23-day period after teaching, six participants gave verbal, one-on-one interviews. Students' solution sheets were used as a basis for discussion. Interviews were structured, with set questions as listed in de Raadt (2007a). The questions were used as a script, but were intended to encourage discussion that was allowed to continue as long as necessary. The questions used were designed not to be leading. Questions were aimed at discovering participants' interpretations of the problem statements, the strategies understood by participants, the articulation of their solutions and their confidence in their solutions.

## 4 Results

A number of results were gained from this experiment. First, data gathered during registration are shown. During the experiment both curricula were delivered to students. The potential to succeed in this delivery was judged by the time used to deliver the more extensive curriculum that explicitly incorporated programming strategies within the schedule. At the end of each of these sessions participants were asked to complete a set of problems that were examined under Goal/Plan Analysis. Finally an inspection of post-course interviews provides deeper insights into the programming strategy potential of the participants after the course.

### 4.1 Data Collected at Registration

The data gathered when participants volunteered for the course are shown in Table 4.1. These data show that the two groups were roughly balanced in gender, age and computing confidence. The two groups differed in responses to computing and web experience self-assessment questions. Experimental group participants showed varying responses to these experience questions. One of the participants indicated they had no previous use of a web browser, even though they used a computer daily. This may have been an error.

**Table 4.1. Demographic, experience and confidence data gathered on registration**

Group	Participant	Gender	Age Group	Computing Experience	Web Experience	Previous Programming	Computing Confidence 1=low to 5=high
<b>Experimental Group</b>	12	male	Less than 25	Daily use	No use	Never	3.0
	21	male	26 – 35	Daily use	Daily use	Some self-taught	4.6
	29	male	26 – 35	Weekly use	Every few days	Never	3.2
	30	female	Less than 25	Daily use	Daily use	Never	4.4
Average							<b>3.8</b>
<b>Control Group</b>	1	male	Less than 25	Daily use	Daily use	Never	3.6
	6	female	Less than 25	Daily use	Daily use	Never	3.5
	13	male	26 – 35	Daily use	Daily use	Never	3.8
Average							<b>3.6</b>

**Table 4.2: Presence of plans and integration for Problem 1**

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Max Initialised					0%				0%	0%
Counter Controlled Loop	Y	Y	Y		75%	Y	Y		67%	71%
Input Plan	Y	Y			50%	Y	Y	Y	100%	71%
Maximum Plan	Y				25%				0%	14%
Output Plan	Y	Y			50%			Y	33%	43%
Input Nested in Counter Controlled Loop	Y	Y			50%	Y			33%	43%
Max Plan Nested in Counter Controlled Loop	Y				25%				0%	14%
Abutment Correct	Y	Y	Y		75%	Y		Y	67%	71%
Overall	88%	63%	25%	0%	<b>44%</b>	50%	25%	38%	<b>28%</b>	<b>41%</b>

**Table 4.3: Presence of plans and integration for Problem 2**

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Sum Initialised	Y				33%	Y	Y		67%	50%
Count Initialised	Y				33%	Y			33%	33%
Sentinel Controlled Input	Y	Y			67%				0%	33%
Sentinel Controlled Count	Y				33%		Y		33%	33%
Sentinel Controlled Sum	Y				33%		Y		33%	33%
Guarded Division					0%				33%	0%
Output Plan	Y	Y			67%	Y	Y	Y	0%	83%
Loop Plans Merged	Y				33%	Y			100%	33%
Inputs Nested in Sentinel Controlled Loop	Y	Y			67%				33%	33%
Output Nested in Guarded Division					0%				0%	0%
Abutment Correct	Y	Y			67%	Y		Y	67%	67%
Overall	82%	36%		0%	<b>39%</b>	45%	36%	18%	<b>33%</b>	<b>36%</b>

**Table 4.4: Presence of plans and integration for Problem 3**

Plan	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Counter Controlled Loop (for Initialisation)	Y			Y	67%				0%	33%
Array Initialisation	Y	Y		Y	100%				0%	50%
Sentinel Controlled Input	Y				33%				0%	17%
Count Set Plan	Y	Y			67%				0%	33%
Counter Controlled Loop (for Output)				Y	33%	Y	Y		67%	50%
Output Plan	Y	Y			67%				0%	33%
Initialisation nested in Counter Controlled Loop	Y			Y	67%				0%	33%
Inputs nested in Sentinel Controlled Loop	Y			Y	67%				0%	33%
Count Set nested in Sentinel Controlled Loop	Y				33%				0%	17%
Output Nested in Counter Controlled Loop					0%				0%	0%
Abutment Correct	Y	Y		Y	100%	Y	Y	Y	100%	100%
Overall	82%	36%		55%	<b>58%</b>	18%	18%	9%	<b>15%</b>	<b>36%</b>

One of the intentions in gathering this data was to exclude volunteers who had completed previous formal study in programming. A number of people signed up for the experiment and were rejected because they had studied programming previously. One participant, identified as Participant 14, who was asked not to attend, came along anyway. The results of this participant are not presented here, but their solutions and transcript are presented in de Raadt (2007a) as some of this participant's responses to interview questions were still of interest. One other participant (21) indicated they had some self-taught programming experience. After discussion with the participant this experience was shown to be a limited amount of HTML writing, which was not seen as significant in this experiment.

## 4.2 Time Load of Explicit Programming Strategy Instruction

During teaching of the curriculum that incorporated explicit programming strategies, added content required additional time to teach, increasing the length of lecture sessions and reducing the time allowed for students to undertake practical work. However, participants undertaking the curriculum with explicit programming strategies were still able to complete the set exercises during the time allocated in the schedule. It was possible for the schedule to be followed in both instances of the curriculum.

### 4.2.1 Goal/Plan Analysis of Participant Solutions

Tables 4.2 to 4.4 show results of the Goal/Plan Analysis for each problem. Several of the solutions presented by novice participants in this experiment contained English language text that described the code the participant would like to have written in their solution when they were not sure how to implement these ideas. Where this was the case, if the text sufficiently described a plan, it was accepted as being present even if it was not described in code. The participants who used text in their code did not create complete or near complete solutions.

Table 4.2 shows the plans present in each participant's solution to Problem 1. The correctness of the integration of the strategies is also recorded and this included correctness of abutment. Unlike experts studied earlier (de Raadt, Toleman, & Watson, 2006), participants in this experiment did not always apply these integration aspects correctly.

The best problem 1 solution was created by Participant 12 from the experimental group who, despite never previously undertaking programming study, was able to produce a well coded solution that was nearly completely correct. This solution, together with those presented by Participant 21, pushed the overall average correctness level for the experimental group above that of the control group despite the abandoned attempt and non-attempt of their group-mates.

One noticeable aspect was the absence of the initialisation of the maximum variable, which was crucial to the Maximum Plan and is required when using JavaScript.

Initialisation was explicitly covered in the curriculum that explicitly included programming strategies. Students undertaking the other curriculum were presented with the opportunity to discover this aspect implicitly. Initialisation was important to the later problems and was applied by a number of participants for those problems. It is not clear why it is absent here.

Table 4.3 shows the strategy correctness of participants' solutions to Problem 2. Participant 29 left after abandoning an attempt at Problem 1, so this participant's solutions were not included in results for this and the next problem.

In this problem again, an outstanding solution was presented by Participant 12 who correctly solved the problem, with the exception of the Guarded Division plan. No participant in either group applied a Guarded Division plan. This suggests that even when it is explicitly incorporated into an introductory programming curriculum, and the consequences of failing to apply the plan are discussed, it is still possible for novice programmers to neglect this particular plan.

This problem was a modified version of the problem given to students in the earlier study (de Raadt, Toleman, & Watson, 2004). Students in the earlier study had completed a semester of instruction under a traditional implicit-only model and achieved an average overall correctness of 57.1% compared to the participants of this experiment who achieved 36%. In the problem statements for Problem 1 and both other problems, students were told they could assume inputs would be valid.

Table 4.4 shows the plan application for the final problem, Problem 3. Again an outstanding solution was presented by Participant 12, who correctly initialised and filled an array to tally user inputs, but failed to output the content of the array using a loop. Participant 30, who did not attempt Problem 1 and presented a confused solution to Problem 2, managed to apply a number of plans for this problem. Participants from the control group showed little ability to demonstrate any of the plans that were needed to solve this problem. This problem is arguably the most complex and, it would appear from these results, it is difficult to implicitly learn the necessary plans required to solve it.

One aspect that was absent in all solutions was the use of a Counter Controlled Loop plan to output the occurrences of numbers. This is not truly surprising as most of the solutions for this problem were incomplete and the only near-complete solution did not apply this particular strategy. Each of the participants from the experimental group applied a counter controlled loop to initialise the array used for tallying.

Table 4.5 shows a comparison of the overall correctness for all problems achieved by each group. There is a

**Table 4.5: Overall plan use by each group**

Overall Plan Use	
Experimental Group	47%
Control Group	28%
All	38%

distinction in overall results for the two groups with the experimental group, who were exposed to a curriculum that incorporated programming strategies explicitly, achieving a greater result.

Participant 12 produced outstanding solutions to each of the problems. It may be that the incorporation of explicit programming strategies suited this participant, who might have performed better than he would have otherwise. One must wonder if this participant would have done as well in the control group and perhaps reversed the results of the experiment.

With the small number of participants in this experiment no statistically significant evidence can be inferred for the superiority of one curriculum over another. These results are useful as basis for the interviews that followed, which allow a deeper and more personal exploration of the participating students' strategy understandings.

### 4.3 Interviews

Following the course, participants were asked to attend an interview. Five of the seven participants and Participant 14 (who had previous programming instruction) volunteered to attend interviews.

Each interview was recorded and transcribed. The transcripts of these interviews are presented in de Raadt (2007a).

From an analysis of the transcripts the following observations are made.

#### 4.3.1 Participants Misinterpreted the Validation Simplification Made to Each Problem

Each problem statement contained the text "Assume the user will enter valid integers only." This additional text was introduced to clarify the problems so no attempt at validation would be necessary. This change was made when these problems were used with expert programmers but for this experiment it may have confused participants rather than simplifying the problems. In interviews participants were asked what this sentence in the problem meant. Three of the five participants misinterpreted this simplification; some suggested validation was necessary because of this statement. No participant attempted to validate inputs.

Other parts of the problem statements seemed to be comprehensible to each participant, even if they did not know how to achieve a solution.

#### 4.3.2 Participants Exhibited Understanding of Plans

As well as demonstrating a higher use of plans in their solutions to problems, experimental group participants verbally described plans, for instance Participant 30 described their application of a Set Counting plan as follows: "After you've put a number that isn't in that range it concludes the program and tells the person what numbers you've put into your little boxes. It goes through zero to nine and it tells you how many are in each box."

Rist (1995) showed that novices can expound and apply plans without explicit instruction of programming strategies. Some control group participants did still learn plans through implicit-only means. In an observable instance Participant 6 stated the following, which could be seen as a description of a Set Counting plan using an array: "I've created an array, because I think that for the program to calculate, between 0 and 9, how many times it occurs, it has to have an array for, say if it's zero, then zero; for one it's one, two three, four... So the array for zero is, like, zero, because arrays start from zero, right? Then, so in the box for zero, say the user enters three times it will refer back to this array zero, it will keep repeating itself in the loop, from then on how many times it gets zero in that box it will get the output."

#### 4.3.3 Participants Failed to Learn Some Plans

It was clear that participants did not learn all the plans they were expected to learn. This was true for participants from the control group who were expected to learn strategies implicitly, for example Participant 6 felt there must be a formula that would take care of the task of calculating maximums: "And probably some formula to determine the highest number (which I don't know how)."

Experimental group participants also failed to learn some plans, even though they had been explicitly exposed to them. For example when Participant 30 was asked how a maximum could be determined, responded, "Can you make the program look at the digits I guess, so you could determine the maximum. I don't know." When Participant 21 was asked, "What does it mean by determine the maximum?", responded with, "Perhaps the maximum sum. I'm not really sure."

#### 4.3.4 Experimental Group Participants Used Plan Terminology and Ideas

On a number of occasions participants from the experimental group (who were exposed to plans and related terminology) referred to parts of their code using the terms used to describe plans or attempted to use plan terminology without specific names.

Participant 12, while discussing the integration of counting with input in Problem 2, said "they have to merge with the loop".

Participant 21, discussing loops in Problem 2, could not remember the terminology for a Sentinel Controlled Loop but described it well: "...and then create a loop... get user input outside and inside so that it's, I can't remember the name." Later Participant 21, while interpreting part of the problem statement, recalled the correct terms and said "Which I did recognise as a sentinel loop."

The use of Goal/Plan terminology was not universal by any means. Participants from the experimental group still resorted to syntactical description when describing their code and needed to be prompted further to elicit possible strategy understandings. Participant 12, who delivered perhaps the best result, stated the following syntactical reading of code: "It's a loop, for loop. For counter equals

zero. Start from zero again. And counter smaller than numberNum. Counter++. And the message is numArray[counter] equals zero.”

#### 4.3.5 Experimental Group Participants showed Confidence in Solutions

One clear finding was that experimental group participants were confident in their solutions, or the ability to correct their solutions if given the chance. This is despite the fact that no participant created a fully correct solution to any of the problems. Participant 21 was confident about all his solutions, even though they were flawed. Participant 30 showed confidence in most of her attempted solutions even though they were flawed; when asked “Does your solution solve the problem?”, replied, “...Well my solution in my head did, not like the first one, so yes. I did understand this question so I could go through the steps of doing it.”

Participant 12, who was the closest of all participants to solving the problems correctly, was realistic about the correctness of his solution. During discussion Participant 12 saw the flaws in two of his three solutions. Interestingly this participant explains his confidence in one of his problems as being the result of understanding the required strategy: “I’m very confident in doing this question because I know the right way to structure [it].”

#### 4.3.6 Control Group Participants showed a Lack of Confidence

When asked if they believed if their solutions correctly solved each problem, members of the control group almost universally showed a lack of confidence in the solutions they had created.

Participant 1 lacks confidence in all solutions except for Problem 2 solution where he claims more time was needed, even though time was not restricted during the test. When this participant was asked, “Does your solution solve the problem?”, answered, “Probably, if I got time to add up more things.” This same participant later describes a lack of confidence in their general programming ability: “I’ll probably mess it up anyways, because I’m still not sure how to...”, and later expresses a typical gap between design and implementation where plans can be applied: “I understand the question. I was thinking through. I got everything right in my head. I just can’t put it onto codes.”

The other control group participant interviewed, Participant 6, showed some confidence in one solution, believing, correctly, that the remaining solutions were flawed.

### 5 Conclusions

The research questions posed earlier are answered by the results of this experiment and the observations of the experimenter/instructor in conducting the experiment.

### 5.1 Explicitly Incorporating Programming Strategies

*Can programming strategies be explicitly incorporated into an introductory programming curriculum?*

Programming strategies can be explicitly incorporated into an introductory programming curriculum. The curriculum used in this experiment is evidence that this can be done.

### 5.2 Balance of Lectures and Practice

*What is the significance of the time consumed by this additional instruction?*

As stated in section 4.2 the additional instruction in the curriculum incorporating programming strategies explicitly did require more time in lecture sessions, but students were still able to complete set exercises by the end of each session. It can therefore be asserted that this additional instruction is balanced by an eased burden on students in completing practical exercises.

This result is useful for our comparison of the curricula, however in regular teaching, lectures and practicals are usually conducted in disjoint time slots; so extending the length of a lecture would not normally impact on practice time.

Having more material in one curriculum over another would increase the burden on student learning with more content to process. This needs to be compared with the effort a student would have to make to develop the needed programming strategies in an implicit-only model.

### 5.3 Assessment of Programming Strategies

*Can programming strategies explicitly taught in an introductory programming course be assessed?*

Goal/Plan Analysis of students’ solutions is far from new, but it is novel as a means of assessment in a programming course. This experiment showed that programming strategies applied to create solutions can be assessed using Goal/Plan Analysis. A limitation of using Goal/Plan Analysis is that it requires students to generate code before it can be assessed. In early stages, assessing generated code might not be the best method of assessing programming strategies.

### 5.4 Impact on Problem Solving Ability

*What impact does explicit strategy instruction have on students and their problem solving ability when compared to an implicit-only approach?*

Through the results shown from Goal/Plan Analysis of participants’ solutions and through interviews it appeared that students exposed to a curriculum that incorporated programming strategies explicitly were more likely to understand and apply those strategies than participants who were expected to learn these strategies implicitly.

It was, by no means, guaranteed that participants explicitly shown programming strategies would understand and apply all of these strategies. It was also

demonstrated that participants exposed to an implicit-only curriculum can learn programming strategies.

## 5.5 Other Observed Effects

*Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?*

Two other observations can be made from the results shown. These are presented in the following subsections.

### 5.5.1 A Vocabulary for Programming Strategies

Some participants in the experimental group, who were exposed to plan terminology during their instruction, went on to use this terminology during interviews. If this were applied during an ordinary teaching period with multiple weeks of instruction and assessment, it would be beneficial to have students able use a common vocabulary of terms. Instructors would be able to describe the strategies they expect students to apply in tasks. It would be possible to allocate marks for the application of specified strategies. Students would have the potential to describe and analyse code using such terminology.

### 5.5.2 Confidence in Solutions

A clear contrast is shown in the confidence participants had in their solutions. Participants from the experimental group, who had been exposed to programming strategies explicitly, were confident about the solutions they presented and the understanding of the strategies needed to complete the solutions. Participants from the control group were not so confident. It is not necessarily clear why this is the case. Perhaps because experimental group participants had been exposed to a higher level of programming thought, they may feel that the underlying syntactical implementation is less difficult to achieve. Reber (1993) suggests that students exposed to implicit-only instruction can gain aptitude but fail to gain understanding of underlying systems. This seems to be consistent with the experience of participants exposed to implicit-only instruction of programming strategies in this experiment who were, in some instances, able to produce partial solutions, but appeared to have a general lack of understanding for programming strategies and the programming processes needed to solve the problems presented.

## 5.6 Flaws in the Experimental Approach

A number of flaws in the experimental approach were realised during and after the experiment.

### 5.6.1 Size of Groups

The size of the experimental and control groups was sufficient to test the potential to incorporate explicit programming strategy content into an introductory programming curriculum and the timing of that incorporation. It was sufficient to allow a small number of participants to experience these curricula and be interviewed on their understandings that may have

developed through this participation in interviews that followed.

Although the Goal/Plan Analysis of participants' solutions showed differences between the groups, the number of participants was too low to statistically infer the superiority of the experimental curriculum. It is not clear that increasing the size of the participant population would produce consistent reproducible results, which appears to be the bane of many explorations in educational settings (Hirsch, 2002).

### 5.6.2 Absorbing Concepts Rapidly

Participants in the study were diligent students. All students were able to follow the course materials and achieve results in paper exercises and practical computer tasks. However, expecting completely correct solutions in the final assessment, which involved generation of code for novel problems, appears to have been more than could be expected from students at the end of two days of instruction. Although exercises were given to reinforce the concepts covered, these may not have been as effective as if they were completed days or weeks later.

The result of this experiment shows that the strategy ability of participants exposed to the experimental curriculum produced an average overall correctness of 39% for Problem 2 compared to students who had been exposed to a semester long, traditional introductory course in programming, who achieved an average overall correctness of 57% on effectively the same problem.

### 5.6.3 Generation of Code can be a Poor Measure

The final assessment asked students to generate code to novel problems, the solutions to which should involve the strategies they had learned in the preceding day and a half. Most of the participants were unable to create complete solutions to these problems. This may be attributable to a lag between

1. exposure to a programming strategy,
2. the ability to comprehend that strategy, and eventually
3. the ability to generate an implementation that applies that strategy.

In this case asking participants to generate code at that stage may have been less effective than gauging their programming strategy skill levels by other means, such as comprehension tests or cases involving errors.

## 5.7 Implications and Future Work

This experiment showed that it is possible to create a curriculum that explicitly incorporates sub-algorithmic programming strategies. The incorporation of such additional instruction does not create an unfeasible burden of time.

There were also noticeable effects on the students participating in the experiment and exposed to this additional instruction. Participants who covered the experimental curriculum appeared more likely to

understand and apply the programming strategies they had been exposed to. These students used terms from a programming strategy vocabulary presented in the curriculum, which could be useful in teaching and assessment if applied to a full scale course. Participants who covered the experimental curriculum claimed confidence in the solutions they had created and their understanding of the strategies used to create them, while students not exposed to this curriculum doubted their abilities.

Some instructors may see these outcomes as encouraging enough to adopt teaching of programming strategies in an explicit manner in full introductory programming courses. An evaluation of a real course with explicitly incorporated programming strategies is planned.

Goal/Plan Analysis is a basic tool for analysing student code and detecting deficiencies in student understanding and, in turn, possible weaknesses in curricula. It has been used here to measure student solutions and as a basis for a deeper exploration of novice understanding. But it appears that its use in this experiment, and in the past, is limited and would not be fully appropriate to assess students at all stages of a full introductory programming course. Multiple forms of assessment are needed to go beyond Goal/Plan Analysis in order to accurately and consistently measure a student's strategy skill during and at the conclusion of a course in introductory programming. Assigning marks to use of strategies in assessments will hopefully encourage students to value this component of the curriculum, devoting study time to programming strategies.

## 6 References

- Cretchley, P. (2006): Does computer confidence relate to levels of achievement in ICT-enriched learning models? In *Education and Information Technologies*. New York, USA: Springer.
- Davies, S. P. (1993): Models and theories of programming strategy. *International Journal of Man-Machine Studies*, **39**(2):237 - 267.
- Incorporating Strategies Explicitly into Curricula (Working Paper), de Raadt, M. <http://www.sci.usq.edu.au/research/workingpapers/sc-mc-0705.ps>. Accessed May 29 2007.
- de Raadt, M. (2007b): A Review of Australasian Investigations into Problem-Solving and the Novice Programmer. *Computer Science Education*, **17**(3):201 - 213.
- de Raadt, M., Toleman, M., & Watson, R. (2004): Training strategic problem solvers. *ACM SIGCSE Bulletin*, **36**(2):48 - 51.
- de Raadt, M., Toleman, M., & Watson, R. (2006): Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications*, **28**(5):55 - 62.
- Hirsch, E. D., Jr. (2002): Classroom Research and Cargo Cults. *Policy Review*, **115**:51 - 69.
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, **36**(4):119 - 150.
- Reber, A. S. (1993): *Implicit Learning and Tacit Knowledge*. New York, USA: Oxford University Press.
- Rist, R. S. (1995): Program Structure and Design. *Cognitive Science*, **19**:507 - 562.
- Robins, A., Rountree, J., & Rountree, N. (2003): Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, **13**(2):137 - 173.
- Soloway, E. (1985): From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research*, **1**(2):157-172.
- Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9):850 - 858.
- Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A., et al. (Year): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proc. Proceedings of the Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia 52:243 - 252.

# An Evaluation of Electronic Individual Peer Assessment in an Introductory Programming Course

Michael de Raadt, David Lai, Richard Watson

Dept. Mathematics and Computing  
University of Southern Queensland  
Toowoomba, Queensland, 4350, Australia

{deraadt, lai, rwatson}@usq.edu.au

## Abstract

Peer learning is a powerful pedagogical practice delivering improved outcomes over conventional teacher-student interactions while offering marking relief to instructors. Peer review enables learning by requiring students to evaluate the work of others. PRAISE is an on-line peer-review system that facilitates anonymous review and delivers prompt feedback from multiple sources. This study is an evaluation of the use of PRAISE in an introductory programming course. Use of the system is examined and attitudes of novice programmers towards the use of peer review are compared to those of students from other disciplines, raising a number of interesting issues. Recommendations are made to introductory programming instructors who may be considering peer review in assignments.

**Keywords:** Introductory programming, assessment, peer review.

## 1 Introduction

Peer learning offers the opportunity for students to teach and learn from each other, providing a learning experience that is qualitatively different from usual student-teacher interactions (Saunders, 1992). Evaluation is a higher-order thinking activity (Anderson et al., 2001). Peer review encourages students to evaluate the work of others and reflect on their own work. Combined with other forms of online communication, peer review can encourage a community of learning, reducing student isolation and further encouraging higher-order thinking (Brook & Oliver, 2003). Peer review can shift instructor workload from marking to other teaching activities (de Raadt, Toleman, & Watson, 2006). Peer review used in regular assignments can increase student retention (de Raadt, Loch, & Addie, 2006).

Peer review can occur in different ways: in person or electronically, between individuals or within teams, over a period or for a single task. Much peer-review literature relates

to assessing peers on contribution to work completed in groups. In online peer-review research, focus is often on online discussion, with involvement in discussion used as a means of assessment (Prins, Sluijsmans, Kirschner, & Strijbos, 2005). The system used in this study, referred to as PRAISE, creates new peer-review relationships between individuals for each assessment item. Reviews focus on student-submitted documents which are provided anonymously (double-blind) to peers for review.

This study evaluates the use of PRAISE in an introductory programming course. Student attitudes to using the system have been measured. Use of the system by novice programmers is described from system statistics. Aspects of implementing PRAISE for an introductory programming course are discussed. Each of these aspects is compared to previous evaluations where PRAISE was used in other disciplines.

This paper begins with a look at available peer-review systems. The PRAISE system is then described. In section 3 the method for evaluating the use of PRAISE is given. Results of this evaluation are shown in section 4. Other evaluations of peer review in computing science are shown in section 5 and related to the findings of this study. Finally, conclusions and recommendations are given in section 6.

## 2 Peer-review Systems

A number of peer-review systems are available. In this study we are most interested in systems that facilitate peer-to-peer evaluations of submitted documents, specifically programming assignments. The following sub-sections briefly introduce existing systems and compare them with PRAISE, the system used in this study.

### 2.1 Existing Peer Assessment Systems

A number of systems share commonalities with PRAISE (Chapman, 2006; Davies & Berrow, 1998; Hamer, Kell, & Spence, 2007; Kurhila, Miettinen, Nokelainen, Floreen, & Tirri, 2003). Examples include CPR, Aropä, and the Moodle Workshop Module.

The Calibrated Peer Review (CRP) system (Chapman, 2006) facilitates submission and review of essays. CPR requires students to undergo training to *calibrate* the peer reviews they later produce. Peer reviews created under CPR are subjective; by comparison PRAISE facilitates submission and review of documents in any format. PRAISE uses objective criteria and instructor moderation to ensure validity of marks without training students.

Aropä is a web-based peer assessment support tool that has been used in a range of academic disciplines (Hamer, Kell, &

Spence, 2007). Aropä allows students to upload documents of any format. Reviews are allocated manually or automatically by an instructor, following which students return to the system and are asked to give quantitative and qualitative feedback on a peer's submission. Quantitative feedback is governed by a flexible marking rubric. Reviews themselves can be subject to 'review' by instructors. Students are awarded marks based on an average of all peer reviews, with weightings given to reviews by instructors. PRAISE uses a similar system for submission and review but combines these two activities into a single step to minimise the number of visits required by students and eliminate complications of multiple deadlines. PRAISE aims at consensus from reviewers on objective binary criteria in order to determine marks. Where consensus is not reached, an instructor moderates the student's submission, overruling previous reviews.

The Moodle Workshop module allows students to submit any electronic document. Reviews can be allocated to students on an automatic basis. Reviews are based on a flexible marking rubric. Comments made by instructors can be saved and shared. The Moodle workshop module has multiple deadlines and does not allow for student flagging or moderation tracking (see section 2.2.2). Unfortunately this Moodle module has not been well maintained and is in a state of disuse within the Moodle community.

An automated peer-review add-on for the Coursemarker Programming Environment was described by Lewis and Davies (2004). Peer review can be combined with automatic assessment on a series of assignments. Peers select appropriate comments from a list; each comment carries a positive or negative mark which is awarded to the submitting student.

## 2.2 Description of the Peer-review System Used – PRAISE

PRAISE stands for Peer Review Assignments Increase Student Experience. Since its inception in 2004, PRAISE has been used in a computing concepts course offered to around 1000 non-computing students per year, a Masters level technology management course with approximately 140 enrolments annually, an introductory accounting course with 230 students, and a professional nursing course with 250 students. A modified version of PRAISE called SQLify is being developed for database courses with an emphasis on SQL query writing (Dekeyser, de Raadt, & Lee, 2007). PRAISE was first used with an introductory programming course in the second half of 2006.

PRAISE delivers rapid feedback to students from multiple sources. Details of the PRAISE system have been described previously with evaluations (de Raadt, Loch, & Addie, 2006;

Student	Instructor
	1. Create assignment instructions 2. Set review criteria
Beginning of Semester	
a. Create document b. Submit document c. Conduct reviews x 2 d. Receive peer feedback	3. Monitor Student Activity
Assignment Deadline	
e. Receive instructor feedback* f. Receive mark	4. Moderate reviews 5. Release Marks

Figure 1. Student interaction with PRAISE

Figure 2a. Submission interface

Figure 2b. Students check criteria and enter a comment

Figure 2c. Instructor's view of submissions

de Raadt, Toleman, & Watson, 2005, 2006). A brief description of PRAISE is given here to provide a context for the findings of this study.

### 2.2.1 Process followed by a Student

For each assignment a student follows a process as described in Figure 1. Students read the assignment instructions and prepare their submissions as they would under a traditional assessment process. They may refer to the marking criteria

which are available prior to submission. When students have completed their document they submit it to the system (see Figure 2a). For the introductory programming course that is the focus of this study, source code files are submitted. The system verifies that the submitted file meets instructor-specified conditions such as type, size and content, and a receipt is emailed to the student.

Initially there is a pooling of submissions, but when this reaches a specified size (around 4-5) the system will begin to allocate reviews to students immediately as they submit their assignment. Students are then directed to complete reviews. The first students to submit must wait until the system notifies them by email to begin reviewing. A single submit-review step allows students to give reflective feedback immediately after submission and reduces the delay from submission to feedback receipt.

Students review the submissions of two peers and are rewarded with marks for undertaking reviews. For each review, students must download and open their peer's submitted document. Students complete a review by checking each criterion against their peer's submission (Figure 2b). Each criterion has a checkbox which is ticked if the peer has fulfilled the criterion. Criteria are phrased in a clear, objective fashion, so that students can review accurately even if they have failed to correctly achieve the criteria themselves. Criteria focus on completion of tasks rather than asking for a judgment of quality; this reduces ambiguity and increases consistency among reviewers. Students must give a comment; they are asked to give praise or positive suggestions for improvement. Students repeat this for each of the two reviews they conduct.

When students have completed reviews they wait to receive feedback from peers. An email is sent to students when their work has been reviewed by a peer. In their own time the students can view feedback on the system. Reviews are shown to students in the same web form used when they conduct reviews, but with controls disabled. Students do not know the value of each criterion and they will not see their overall assignment mark until it has been released by an instructor.

## 2.2.2 Process followed by an Instructor

Instructors follow a process for each assignment as shown on the right side of Figure 1. Before the semester begins the instructor must create the assignment. A key goal when using PRAISE is clear, objective criteria, focusing on completion of tasks set in the assignment instructions. Criteria should encourage consistency between reviewers. The criteria are stored in the system. Once this is done, students can begin the course and start completing assignments. Students can submit assignments at any time after the start of the course. Up to the assignment deadline the instructor will monitor the submission process but is not required to intervene.

Moderation of student reviews is achieved using the interface shown in Figure 2c. This interface shows a list of submissions for a particular assignment, each row relating to one student's submission. Instructors have access to each student's number, name, email address, submitted file, submission date, time and file size, a log of the submission details, the reviews conducted by the submitting student and peer reviews of the student's submission. Relationships between reviewer and reviewee are highlighted when the mouse pointer is moved over a review icon. The system attempts to consolidate reviews of the

student's submission. If the submission has been reviewed twice and reviewers agree according to the criteria, the system will suggest a mark based on the value of each criterion. If reviews do not agree, the system will highlight the submission for instructor moderation. Past use of the system (de Raadt, Toleman, & Watson, 2005) indicates that the instructor will conduct moderations on roughly 50% of submissions depending on the complexity of the criteria; this means the instructor will accept a mark suggested by the PRAISE system, based on peer reviews, for 50% of submissions. This can allow time that would normally be spent marking to be used for other teaching activities. The instructor uses the same form that students use when conducting reviews. Students are notified by email when an instructor moderates their submission and the moderation appears with other reviews on the Marks and Reviews page.

When all submissions requiring moderation have been attended to and all conflicts are resolved, the instructor releases marks for all submissions of the assignment. Students are sent an email and can check their marks on the system.

## 2.2.3 Features

PRAISE boasts a number of features not available in other peer-review systems.

- **Single submit-review step**

PRAISE can arrange new peer-review relationships for each assignment without instructor involvement. This is a big time-saver for instructors. This also benefits students. Only a single deadline is needed for both submission and review. Students are not required to return to the site for the sole purpose of completing reviews. Most students can immediately undertake reflection and evaluation on activities they have just completed. Waiting time to receipt of feedback is reduced. By allowing reviewing immediately after submission, students can work ahead in the course. In previous use of PRAISE some students have finished all the assignments of a course in the first few weeks. As students review previously submitted documents it is also easy to accommodate students submitting after the deadline. PRAISE applies late penalties automatically but late students can still complete reviews.

- **Practice submission**

PRAISE allows only a single submission for each assignment. This can create anxiety in students unsure about using the system. To counter this, PRAISE can be set up with a 'practice' assignment allowing students to experience submission and review (with instructor-created documents to review).

- **Flagging**

Even though instructors moderate assignments, some students are uncomfortable when peer reviews are used as a basis for creating marks. PRAISE allows students to flag peer reviews they believe are inaccurate. When a peer review is flagged an instructor must perform moderation on that student's submission.

- **Tracking moderations**

An instructor can choose to award marks based on peer reviews when there is no conflict. Under this scheme, a top student who produces good work will consistently receive good peer reviews and may never receive a

moderation review from an instructor. If this is the case the student may feel they are not receiving the level of attention they deserve from the instructor. PRAISE counts instructor moderations for each student through the course. Targets can be set; for instance, “after assignment four all students have been moderated at least twice by an instructor.” If the moderation count is below target, the instructor will conduct a moderation review, even if both peer reviews are consistent.

### 3 Evaluating Peer Review in an Introductory Programming Course

The following questions were used to guide the evaluation of peer review and of the PRAISE system in the context of an introductory programming course. An introductory programming instructor may ask these questions when considering adoption of peer review in their course.

- RQ1. Can peer review be applied to assignments in an introductory programming course and what are the logistical differences when compared with a traditional submission model?*
- RQ2. Do novice programmers find PRAISE easy to use?*
- RQ3. Do novice programmers appreciate the learning benefits of undertaking peer review?*
- RQ4. Do novice programmers value reviews of their work by peers?*
- RQ5. Is there significant marking relief when using peer review compared to marking paper-based programming assignments?*

Answers to these questions are considered in section 6.1 of the Conclusions.

#### 3.1 Methodology

The use of PRAISE in an introductory programming course was evaluated in two ways. A survey, designed to elicit student attitudes towards the system, was conducted at two points during the course. Also, statistics on the use of the system by students were gathered from data stored in the system. This evaluation took place during the second semester (in the second half of the year) in 2007.

#### 3.2 Setting – The Course<sup>1</sup>

The focus of this study is the use of peer review in an introductory programming course at the University of Southern Queensland. The course uses the language C in a procedural paradigm with a focus on syntax and sub-algorithmic problem-solving strategies.

Students are enrolled in the course in either *on-campus* mode or *external* mode. These two modes are distinguished by attendance, with on-campus students attending lectures, tutorials and practical classes. External students may be studying anywhere in the world. Based on first assignment

submissions, 28% of students are enrolled on-campus and 72% externally.

There are six assignments in the course, each requiring the novice programmer to generate a source code file containing a problem solution. Each assignment contributes 8 marks to the final assessment; the remaining 52 marks are allocated to the end-of-course examination. Within each assignment 6 marks are allocated to the quality of the student’s submission as judged through peer reviews and instructor moderation. A further 2 marks are awarded for completing two reviews (one mark per review). To evaluate code submitted by peers, students are asked to compile, run and test the solutions while checking the review criteria. This form of testing, as part of reviews, has not formerly been used with PRAISE.

Assignment deadlines are regular, roughly two weeks apart. Assignment deadlines occur at midnight on the due date. After this, late penalties are applied to encourage students to stay on track. Smaller, regular assignments are used to encourage continuous involvement in the course. Students must complete one assignment before they can move onto the next. Regular assignments allow easy identification of students falling behind, who might require intervention.

Support mechanisms provided to students include online forums, email, phone and personal contact with instructors. Students are encouraged to make use of the support mechanisms in that order unless personal matters arise. The forums are monitored on a regular basis.

Information was provided to students explaining why peer review is used in the course. Students were able to read a justification for using the system, view a short video of how to use PRAISE and try out the system through a practice assignment.

#### 3.3 Survey

Two surveys were conducted during the semester. Students were able to complete the first survey after finishing the first assignment and the second after the sixth (and final) assignment. The surveys were conducted online using a web form. The questions used in the survey were drawn primarily from previous evaluations of the system (de Raadt, Toleman, & Watson, 2005, 2006). Questions used the statements in Table 1. In the second instance of the survey, one question was dropped (question 2) and several were added. Questions used with each survey are marked with a tick in the survey column of Table 1.

Questions asked in the first survey at the beginning of the semester (after the first assignment) were phrased in the present tense. Questions in the second survey asked at the end of the course reflected back on the use of the system using the past tense. For instance, “there is support” was later phrased “there was support” and “seems easy to follow” was later “seemed easy to follow”. The subject of each question was the same in both surveys.

The questions focused on the course, the assignments used in the course, and, of primary concern in this study, students’ attitudes towards peer reviewing. Students were asked for their agreement with the statements in Table 1 and responses were captured using a five-point Likert scale with possible responses *Strongly Disagree*, *Disagree*, *Neutral*, *Agree* and *Strongly Agree*. Negatively phrased statements are marked with an asterisk (\*).

<sup>1</sup> The term course is used to refer to a single semester-long period of study. This may be equivalent to a subject, unit or paper in other institutions.

**Table 1. Survey questions for both surveys**

Question	Survey 1	Survey 2
1 I feel confident that I will pass this course.	✓	✓
2 This course is important to my degree program.	✓	
3 I enjoyed the challenge of completing programming activities in this course.	✓	✓
4 The assignments were big and took a lot of time to complete.*	✓	✓
5 There was support if I got stuck when completing assignments or reviews.	✓	✓
6 The process of submitting assignments was easy to follow.	✓	✓
7 The process of completing reviews was easy to follow.	✓	✓
8 I felt limited by only being able to submit each assignment once.*		✓
9 Submitting assignments electronically requires less effort than submitting an assignment on paper.		✓
10 Completing regular assignments forced me into a regular pattern of study.		✓
11 Reviewing other's work helped me understand the concepts covered in each assignment.	✓	✓
12 Seeing the work of others showed me different ways to complete tasks.	✓	✓
13 I would rather receive marks from instructors only.*	✓	✓
14 Interacting with peers through reviewing motivated me to produce better assignments.	✓	✓
15 Communicating with peers through reviewing gave me the sense I was not alone in my studies.	✓	✓
16 I was uncomfortable that others saw my work.*	✓	✓
17 When I saw other students' submissions I compared them to my own work.		✓
18 The feedback I received from my peers through reviews was useful to me.		✓
19 Feedback on my submissions came rapidly from peers and instructors.		✓
20 The quality of feedback from peers and instructors was as good or better than what I would expect on paper based assignments marked by hand.		✓
21 I would be happy to use the same submission and review facilities in other courses.		✓

### 3.4 Usage Statistics

A number of statistical measurements of the system were achieved by analysing data available in the system. The aspects measured were as follows.

- Time from submission to deadline
- Time from submission to receipt of first review
- Proportion of moderations required due to conflicts
- Use of flagging by students

## 4 Results and Discussion

This section discusses the results of the evaluation. First the survey participants' responses are described. Following this, usage statistics are shown. Finally, the results are compared with previous evaluations of the use of PRAISE.

### 4.1 Survey Responses

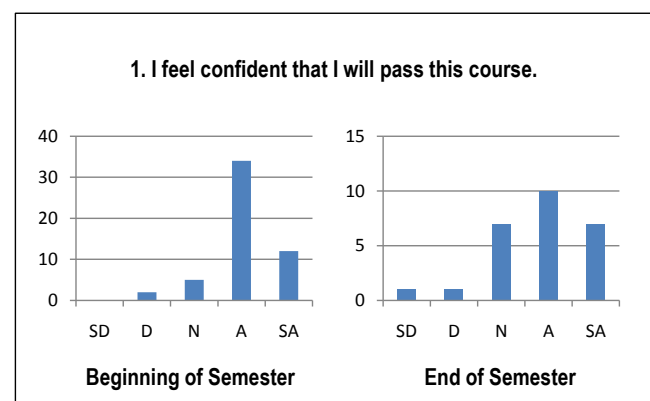
Table 2 shows response rates for the two surveys. Participants include males and females, school leavers and mature-age students and part-time and full-time students. The proportions for these aspects were not captured as part of the survey.

**Table 2. Survey response rates**

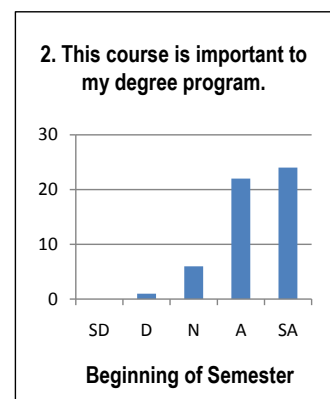
	Submissions	Surveys	Response Rate
Asst. 1/Survey 1	79	53	67%
Asst. 6/Survey 2	38	26	68%

There were 14 participants who responded to both the first and second surveys. The responses to each survey are considered independently rather than as a continuous change of attitude. Responses are grouped by the focus areas: course, assignments and reviewing.

#### 4.1.1 Questions about the Course

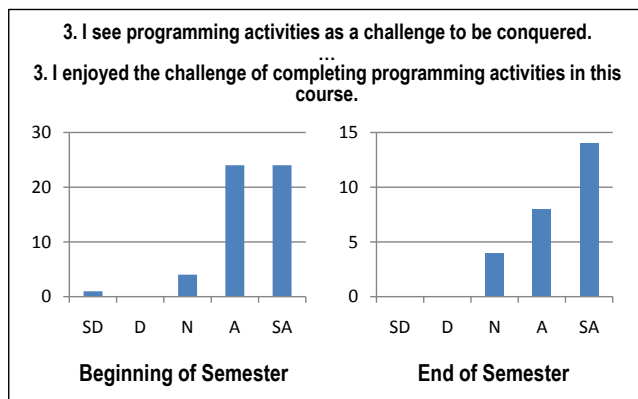


When asked if they were confident about passing the course almost all students showed confidence at the beginning of the semester (question 1 beg: SD+D=4%, N=9%, A+SA=87%).



Closer to the end of the semester, students were predominantly confident, but some gave a neutral response (question 1 end: SD+D=8%, N=27%, A+SA=65%).

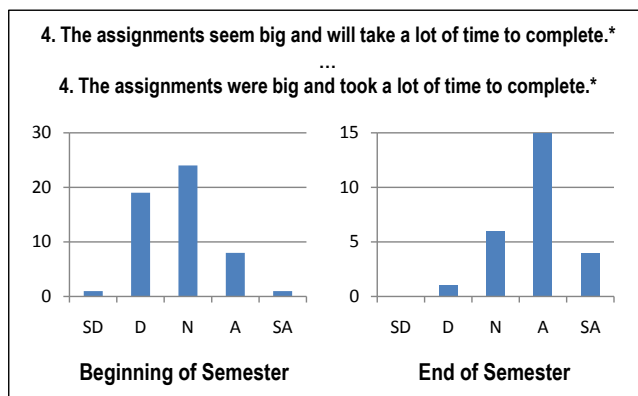
At the beginning of the semester, most students suggested the course was important to their studies (question 2 beg: SD+D=2%, N=11%, A+SA=87%).



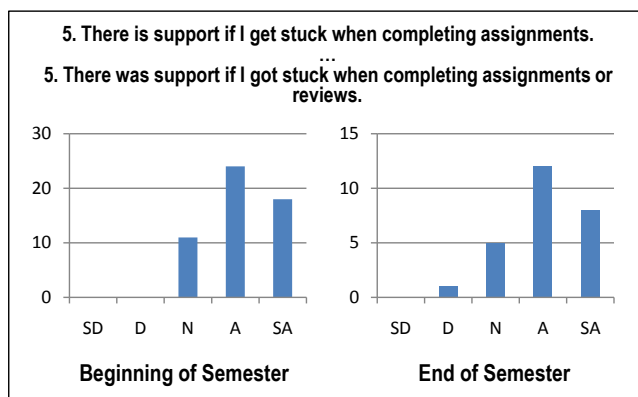
At the beginning of the semester, most students agreed that programming is challenging (question 3 beg: SD+D=2%, N=8%, A+SA=91%). Responses to these questions paint a positive picture for the course. Participating students seem to have good intentions and motivation.

At the end of the course students were asked if they enjoyed the challenge of programming and another strong response was recorded (question 3 end: SD+D=0%, N=15%, A+SA=85%).

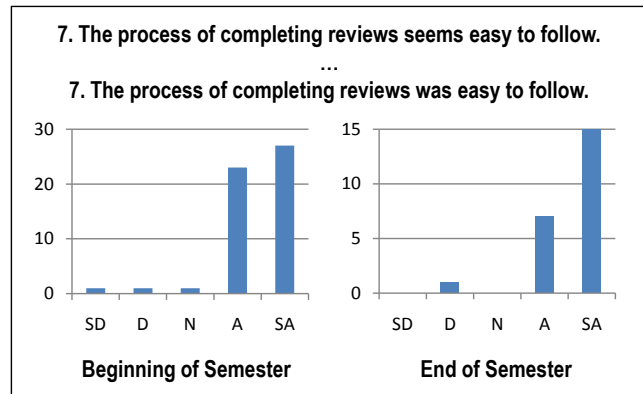
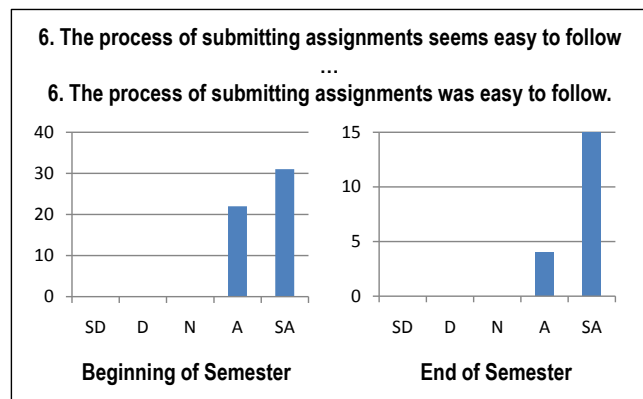
#### 4.1.2 Questions about the Assignments



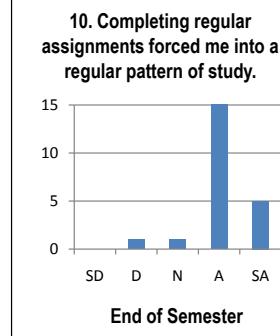
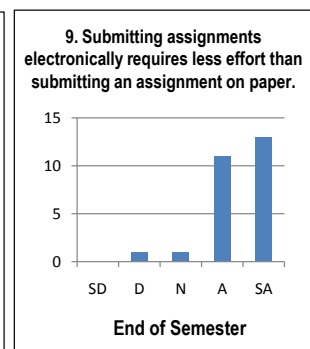
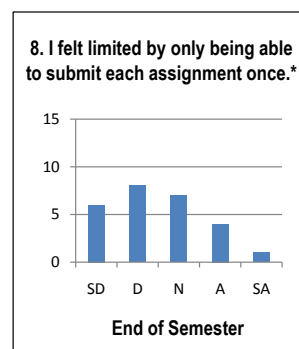
Students were divided over their perceptions of the scale of the assignments at the beginning of the course. Question 4 was phrased negatively and yielded responses (question 4 beg: SD+D=38%, N=45%, A+SA=17%) revealing many neutral participants. At the end of the semester, after doing the work, more students agreed that the assignments were big (question 4 end: SD+D=4%, N=23%, A+SA=73%).



Students seem happy with the apparent level of support as shown from responses to question 5 (beg: SD+D=0%, N=21%, A+SA=79%; end: SD+D=4%, N=19%, A+SA=77%).



Questions 6 and 7 relate to the ease of submission (beg and end: SD+D=0%, N=0%, A+SA=100%) and review (beg: SD=4%, N=2%, A+SA=94%; end: SD=4%, N=0%, A+SA=96%). Both early in the course and at the end, students seem very at ease with both these processes.



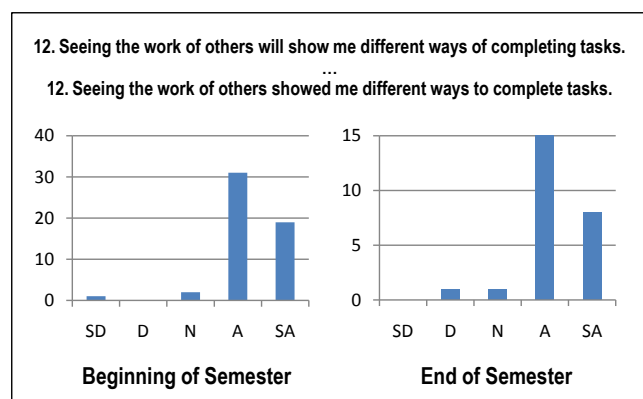
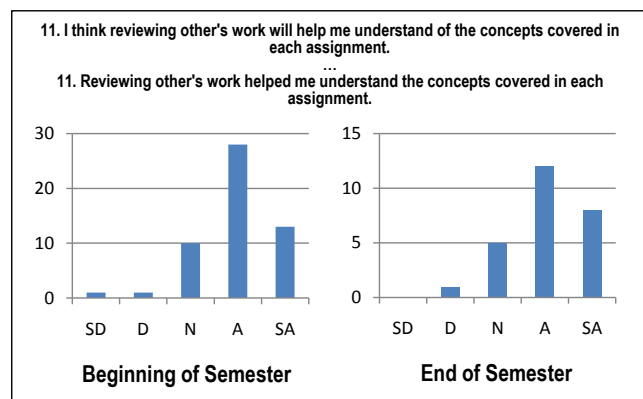
Questions 8, 9 and 10 were asked only at the end of the semester. Question 8 related to the limitation of only being able to submit once for each assignment. This was a negatively phrased question showing responses (question 8 end: SD=54%, N=27%, A+SA=19%). These responses indicate that a majority of students are comfortable with the single submission but there is a large number who are not. Question 9 asks about the ease of submitting an electronic document over a paper submission (question 9 end: SD=4%, N=4%, A+SA=92%). This finding is useful even for instructors using

submission but there is a large number who are not. Question 9 asks about the ease of submitting an electronic document over a paper submission (question 9 end: SD=4%, N=4%, A+SA=92%). This finding is useful even for instructors using

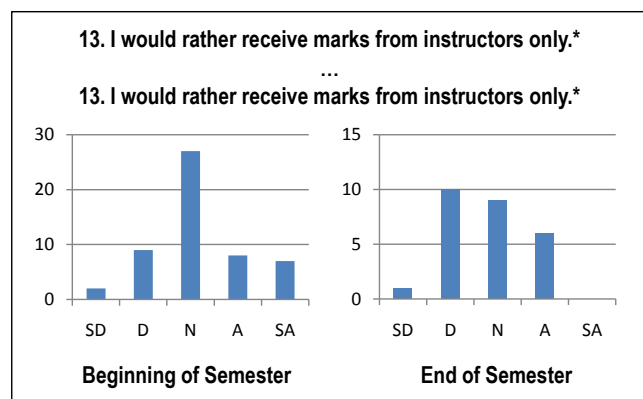
electronic submission without peer review. One of the intentions for having six regular assignments was to maintain regular student involvement in the course. Students agreed that this had been achieved (question 10 end: SD=4%, N=4%, A+SA=92%).

### 4.1.3 Questions about Reviewing

Questions 11 to 21 were designed to discover how students value reviewing as part of their assessment.

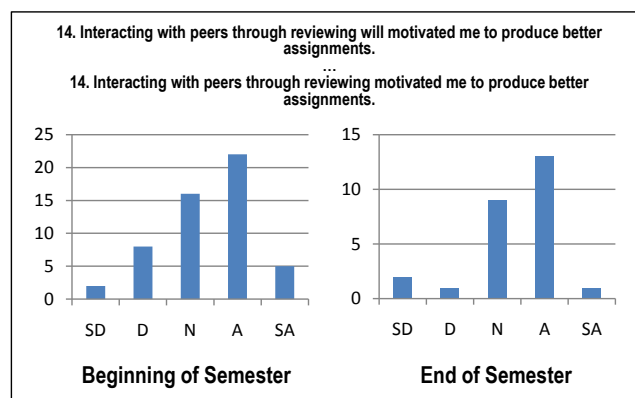


Results for question 11 (beg and end: SD+D=4%, N=19%, A+SA=77%) and question 12 (beg: SD+D=2%, N=4%, A+SA=94%; end: SD+D=4%, N=4%, A+SA=92%) describe the participating students' perceptions of the learning benefits inherent in undertaking peer review. It is clear that students saw these benefits early in the course and at the end. Programming students really appreciate seeing the solutions submitted by their peers.

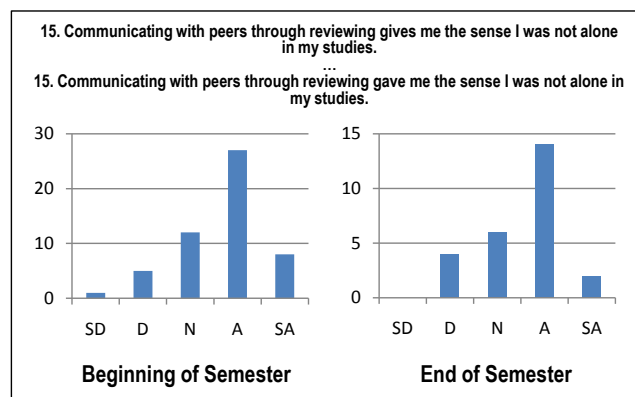


Question 13 puts a value on the use of peer reviews as a means of assessment (question 13 beg: SD+D=21%, N=51%, A+SA=28%; end: SD+D=42%, N=35%, A+SA=23%). This

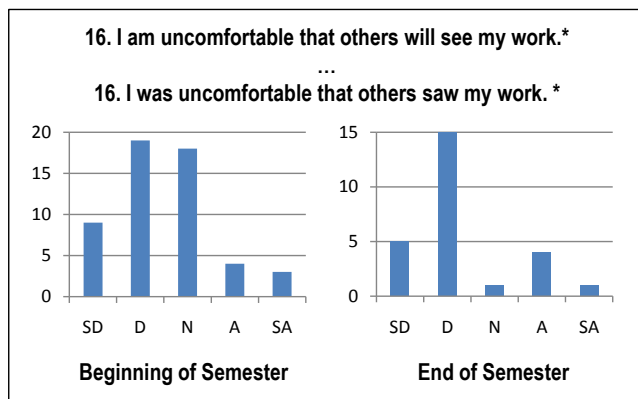
question is phrased negatively. At the beginning of the semester most students were neutral in their response, but it is clear that a good proportion of students want an authoritative instructor awarding marks. At the end of the semester students seemed to value peer-review slightly higher. This implicitly gives a value to the feedback students receive from their peers. It should not be assumed that feedback in peer reviews is valued as highly as instructor feedback.



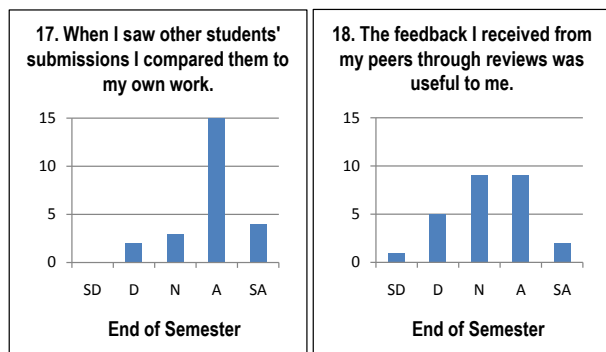
Question 14 measures the motivation of participating students gained by knowing that a peer will see their submission (question 14 beg: SD+D=19%, N=30%, A+SA=51%; end: SD+D=12%, N=35%, A+SA=54%). Many students feel motivated by this (more than in any previous cohort). A few students do not, but this does not necessarily imply that peer reviewing is de-motivating; it may be that participating students who disagreed with this statement are motivated by forces other than their peers seeing their work.



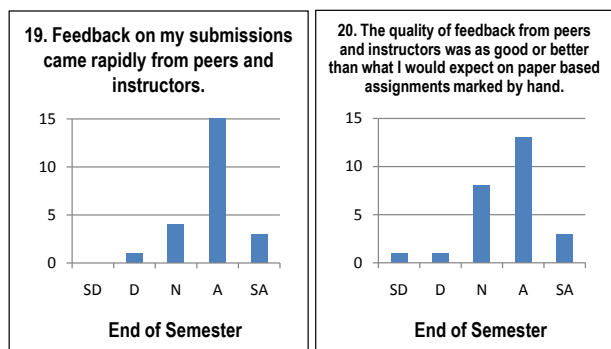
Question 15 measures the sense of community that arises out of peer review (question 15 beg: SD+D=11%, N=23%, A+SA=66%; end: SD+D=15%, N=23%, A+SA=62%). As mentioned earlier, many students in the course are externals who can feel isolated in their studies. It appears that, for most students, peer reviewing encourages a sense of community which, together with online communication, can positively affect learning outcomes.



Question 16 measures the level of comfort with peers viewing a student's submission. The system provides double-blind anonymity in reviews. This question was phrased negatively with early responses (question 16 beg: SD+D=53%, N=34%, A+SA=13%) suggesting students are mostly comfortable or neutral. At the end of the semester most students suggested they were comfortable (question 16 end: SD+D=77%, N=4%, A+SA=19%). Few students are uncomfortable, but this is still an area of concern.



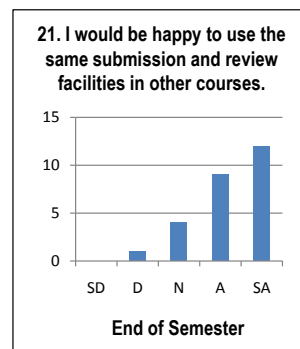
Questions 17 to 21 were asked in the end of semester survey only. Responses to question 17 (end: SD+D=8%, N=12%, A+SA=81%) indicate students had undertaken reflection, which is one of the desired pedagogical benefits of peer review. Responses to question 18 are mixed (question 18 end: SD+D=23%, N=35%, A+SA=42%) and show what has been found in previous surveys from other disciplines: that students do not necessarily value the feedback they receive from peers.



Question 19 describes the students' satisfaction with the speed at which they received feedback and this is quite positive (question 19 end: SD+D=4%, N=15%, A+SA=81%).

Question 20 asks if students are happy with the general quality of feedback they receive through the PRAISE system when compared to paper-based assignments. Students are generally

happy with the quality of feedback they received (question 20 end: SD+D=8%, N=31%, A+SA=62%).



The final question, question 21, asks if students would like to use a system like PRAISE in other courses they are studying. Students were quite happy with the system and would like to see it used elsewhere (question 21 end: SD+D=4%, N=15%, A+SA=81%).

## 4.2 Comments

The free comments made by students were encouraging and predominantly positive. The following are positive comments from the initial survey.

- *I think reviewing other peoples work is great.*
- *...the support available is fantastic.*
- *i like the review system, it works well.*
- *I like the regular assignments and the review system.*
- *I was worried at first that other students would be able to view my work. However, since there are strict guidelines as to how to review someones work and that we are all encouraged to give positive feedback, I felt more comfortable.*

After the initial survey, one student raised a problem unique to using peer review in a programming context where students need to compile and test code. Students are encouraged to use an ANSI standard compiler. Examples are suggested to students and a free compiler is available for download. Students are asked to be aware that peers using other compilers may be reviewing their work. However, there is never complete compatibility between different compilers and how they behave.

- *...when I compiled my assignment through OSX terminal with g++, I got no warnings or errors, but people on windows compiling my source code did.*

Initially, one student stated their refusal to undertake peer reviews.

- *I have great reservations with "peer review", and for myself do not partake due to the possible harm caused. After all what would a student know about the subject they are learning? ... I would prefer information straight from the lecturer as I would trust the source of the information...*

This student left their name with this comment. This was seen as an invitation for a response. The student was encouraged to undertake reviews as a learning activity for their own benefit and assured that the process of reviewing is was overseen by instructors. The student did go back and complete reviews. This comment exemplifies a nervousness about the use of peer review for assessment, which itself is quite novel. Students must be made aware of the justification and learning benefits of peer reviewing before they are involved.

In the second survey responses were still predominantly positive.

- *...set up brilliantly to study externally...*
- *Peer review is a new experience for me, an uncomfortable one at 1st, but it can be of benefit if the student puts in the work to start with*
- *I really liked the fact that there was so much flexibility...*
- *I found the assignments to[o] big but highly beneficial. It allowed me to correct my own mistakes and write better code...*
- *I enjoyed this course and was challenged by it. Mostly I found the comments from peers to be supportive and helpful...*

After experiencing the system over the semester, a number of students raised their concerns about certain aspects of the system, many related to workload in the course.

- *...I often did not know what to say because if their code was not working I had no clue why.*
- *I strongly feel that this unit covers far too much material over a short period of time.*
- *...six assignments may have been a little over the top, however it did make me study more regularly.*
- *The regimented structure of assignment submission dates for this course is hard for students studying and working full-time ... I found myself in the position of attempting assignments without having done the required course work*
- *...only negative i found was review comments weren't particularly useful.*

### 4.3 Usage Statistics

Statistics were gathered regarding timing, reviews and moderation.

#### 4.3.1 Time from submission to deadline

PRAISE allows students to work ahead. Students are made aware of this fact at the start of semester. Some students take advantage of this, others do not, but neither is necessarily preferred. However, measuring how far ahead students are working is an indication of student motivation. Table 3 shows statistics about the time between submission and the deadline. Late submissions are excluded as these may have involved extensions or other complicating factors. The median is the best guide and shows a reduction, with half of the student cohort submitting assignment 1 one day and nine hours before the deadline but only three to six hours before the deadline in the last three assignments.

**Table 3. Time between submission and deadline**

Asst.	Longest (Earliest)	Mean	Median	Shortest (Latest)
1	23days	3days 3hr	1day 9hr	60min
2	13days	1day 17hr	12hr 20min	16min
3	5days	1day 1hr	11hr 9min	2min
4	9days	1day 12hr	6hr 26min	5min
5	2days	8hr	2hr 50min	0min
6	9days	19hr	5hr 42min	3min

#### 4.3.2 Time from submission to receipt of first review

One of the benefits of a single submit-review step is that students can receive reviews from peers shortly after they submit. To measure the effectiveness of this feature the delay between a submission and the first review is captured. These figures also give an indication of the time taken by students to complete reviews. The longest and shortest delays, and the mean and median delays are shown in Table 4.

**Table 4. Time from submission to first review**

Asst	Longest	Mean	Median	Shortest
1	13days	15hr	4hr 8min	11min
2	10days	1day 4hr	2hr 45min	24min
3	3days	8hr	3hr 18min	12min
4	7days	23hr	2hr 9min	10min
5	3days	5hr	1hr 6min	25min
6	9days	20hr	1hr 47min	31min

Again the best guide to measuring the delay for feedback is the median delay. For the first assignment, half of the student cohort received feedback within 4 hours or less. For later assignments this reduced to a little over an hour. These delays are affected by the time between submission and the deadline. When students submit closer to the deadline there is a greater concentration of submissions, so feedback is returned sooner. Students submitting earlier (further from the due date) generally have to wait longer for feedback to arrive. All of these figures, though, are commendable considering that the delay from submission to feedback receipt in a traditional paper-based assessment involving postage can be up to six weeks.

Also of interest in Table 4 is the minimum time between submission and review. Although students were generally receiving feedback faster over the semester, the minimum gap increased in the later assignments, showing that students were taking more time to complete reviews, perhaps due to the greater complexity and number of review criteria.

#### 4.3.3 Proportion of moderations required

The proportion of submissions which require moderation is a measure of the consistency achieved between peer reviewers. This in turn is an indication of the ease with which students were able to apply the review criteria. The rates where moderation was required are shown in Table 5.

**Table 5. Proportion of moderation required**

Assignment 1	61% (100% conducted)
Assignment 2	72%
Assignment 3	80%
Assignment 4	76%
Assignment 5	67%
Assignment 6	73%

For the first assignment all submissions were moderated, even though only 61% of reviews were conflicting. This was done to encourage students early and provide a good example of the reviewing standard expected. It also provided a chance to detect lazy reviewers – students who simply check all criteria without referring to, or testing, the submitted source code. For

later assignments, the number of conflicts, and therefore moderations, increased. It should be noted that there were more criteria used with reviews in these later assignments, which may have increased the likelihood of conflicts.

#### 4.3.4 Proportion of Reviews Flagged

The last measure gathered from use of the system was the proportion of peer reviews flagged by students. If students were unhappy about a review they had the option of flagging it. A flagged review forces an instructor to moderate the submission. The level of flagging for the assignments is shown in Table 6.

**Table 6. Proportion of all peer reviews flagged**

Assignment 1	3%
Assignment 2	4%
Assignment 3	2%
Assignment 4	2%
Assignment 5	1%
Assignment 6	3%

The level of flagging is an indication of the confidence students place in the reviews they receive from their peers. From the survey questions described earlier it is clear that, while students value the experience of undertaking reviews, they do not always have confidence in the feedback they receive from peers. Despite this, the use of flagging was quite low, indicating that students either believe the reviews are accurate or are confident an instructor will correct inaccurate reviews.

#### 4.4 Comparison to Previous Evaluations in non-Programming Courses

Novice programmers find submitting assignments and conducting reviews easy. Their confidence is superior to students from other disciplines in previous evaluations.

Previous evaluations have shown that students do not value reviews from peers as highly as instructor feedback. This attitude is also evident in the current evaluation, with novice programmers valuing peer reviews slightly less than in previous evaluations (see question 13).

Students participating in the current evaluation are more motivated than students in previous evaluations by knowing peers would view their work. Survey participants in previous evaluations were relatively neutral about viewing and evaluating the work of their peers. A clear distinction to previous evaluations is the high value students place on being able to view, test and evaluate the work of peer novices. In introductory programming this appears to be a major attraction.

Students gave enthusiastic comments about seeing others' work and showing off their own work. Some negative attitudes were given in comments. Most negative comments were based on the workload of the course rather than the use of peer review. In previous evaluations it was concluded that many negative attitudes arose from students being ill-informed about the motivation for using peer review and unaware of the benefits to learning outcomes. The response has been to promote peer review and its benefits prior to use. This was done in the current course but perhaps this dissemination could

be improved. Several students felt the assignments were too big.

Rates of moderation were higher than experienced in other disciplines through previous evaluations. This indicates that students are producing less consistent reviews, which is a sign of the quality and complexity of the assignment instructions and criteria. Clear criteria need to be created and refined, which may require several iterations of each assignment.

Previous evaluations found that most students submit on the due date, but in each course where evaluation was undertaken several students would work ahead, some completing all assessments in the first few weeks. This does not seem to be the case in the introductory programming context. Novice programmers submitted closer to the due date and no student worked to submit assignments ahead of schedule, even after they were encouraged to do so.

Novice programmers took more time to produce reviews than has been experienced in other disciplines. In a computing concepts course for non-computing students, the median time from submission to first feedback was 1hr 21min where in the current course the overall median was 2hr 33min. Novice programmers took longer to evaluate the work of their peers. Survey participants indicated that they enjoyed seeing the work of their peers and comparing it to their own.

### 5 Relation to Previous Evaluations of Peer Assessment in Programming

Sitthiworachart and Joy (2004) describe the use of peer review together with automatic marking for a single assignment in an undergraduate programming course. The workings of the system used are not described in detail, however some information is given. For the peer-review component, students are asked to subjectively rate three peers' submissions using set criteria, each associated with a scale of marks. Marks awarded to students are an average of three reviews of their submitted work. In evaluating their system Sitthiworachart and Joy found 65% of students were satisfied with their marks and 51% regarded feedback from peers as useful. Through a combination of attitudinal measures captured in this study it could be argued that student satisfaction with PRAISE is higher, but it is interesting to note that peer feedback was not valued highly in either evaluation. Students expressed a lack of confidence in their marks in comments under the system used by Sitthiworachart and Joy, which caused them to suggest moderation as a means of providing fairer reviews. PRAISE uses instructor moderation, which may be why students showed higher confidence in that system.

A study by Chinn (2005) measured the validity of peer-assessment in an algorithms course. The study found a correlation between marks from peer-reviewed and other activities, suggesting that peer-awarded marks are consistent. Chinn noted that students tend to focus on high-level errors, identifying these more often than low-level errors. Student attitudes towards the validity of peer assessment discovered by this study indicate that novice programmers accept the marks they receive, with relatively low levels of flagging.

### 6 Conclusions

In this section the questions raised in section 3 will be addressed first. This is followed by discussion of differences encountered between use of PRAISE in an introductory

programming course and in other courses. Finally, future work is suggested.

## 6.1 Research Questions

*RQ1. Can peer review be applied to assignments in an introductory programming course and what are the logistical differences when compared with a traditional submission model?*

Peer review fitted the assignments in the introductory programming context nicely. Using simple, fixed criteria it was possible to focus student attention on important syntactical and problem-solving aspects of assignments. Peer review has allowed for smaller, more frequent assignments focused on recent topics.

One difference comes in asking students to undertake testing of their peers' solutions for reviews. Previous use of PRAISE has asked students to undertake relatively passive observations when evaluating the work of their peers.

Anonymity becomes a difficult balancing act when using peer review. In examples shown to novices, comments are written at the start of source code files to identify the author and other relevant details. Such comments are encouraged in the course, but students have to be asked to remove these comments before submitting and many fail to do so. Some aspects of the assignments are designed to allow students to personalise their work, hopefully making the tasks more relevant to them. An example of this occurs in most assignments. One example in the first assignment involves students outputting their name in asterisks. While these aspects of personalisation are pedagogically desirable, they reduce the level of anonymity and potentially the accuracy of reviews if peers are familiar with each other.

Some compatibility issues arose during reviews of assignments. Students are working on different platforms and development environments so while one compiler might not warn a novice to add a blank line at the end of their source code, another compiler will. Asking students to consider the environment where their code will be tested is not bad as it encourages them to write more compatible code and avoid compiler specific tricks.

*RQ2. Do novice programmers find PRAISE easy to use?*

Absolutely, and with more confidence than any previously surveyed cohort of PRAISE users from other disciplines.

*RQ3. Do novice programmers appreciate the learning benefits of undertaking peer review?*

Previous studies have shown that peer review encourages students to become more involved in the course, to feel less isolated, and to move towards higher-order thinking

It appears that novice programmers recognise the benefits of conducting peer reviews. They relish the chance to view, test and evaluate the code of others. Many are motivated to produce better work because of peer review.

*RQ4. Do novice programmers value reviews of their work from peers?*

Novice programmers are quite neutral about whether they would prefer feedback from peers and instructors through reviews or from instructors alone. Some students feel this is beneficial and some feel it could be detrimental. Additional feedback to students should positively improve their learning

outcomes, but even without this extra feedback, the remaining benefits inherent in peer review still make its use worthwhile.

*RQ5. Is there significant marking relief when using peer review compared to marking paper-based programming assignments?*

There is some reduction in the marking of individual assignments. The process of moderation is somewhat quicker than marking code on paper or by other means. The number of submissions marked by an instructor can be reduced by 20-40% in an introductory programming course using peer review as a basis for assessment. This is not as significant as in other disciplines. Perhaps this rate of moderation can be improved by refining review criteria.

There are costs associated with establishing good criteria and managing the system, but then there may be equivalent costs in any submission and marking system.

Based on the answers to these research questions the authors recommend that introductory programming instructors considering adopting peer review to improve learning outcomes for their students.

## 6.2 Comparison with Non-programming Courses

A number of differences were found between the attitudes and practices of novice programmers undertaking peer review and those of students from other disciplines. The following is speculation on why these differences are occurring.

*Why are novice programmers more motivated by peer review?*

Assignments in introductory programming courses are arguably more challenging than in those in other disciplines. Assignments require students to undertake problem solving, and solutions are students' expressions of their development in programming expertise. Peer review gives novice programmers an opportunity to showcase their achievements.

*Why don't novice programmers work ahead?*

It seems likely that novice programmers would work ahead if they could. It may be they are prevented from doing so by the cumulative nature of materials which build up over the course of study. It may be that programming concepts require longer to absorb. It may be that assignments are more challenging than assessments in other disciplines, taking longer to produce submissions. Then again it may be that novice programmers, or perhaps just this cohort, are less motivated to work ahead.

*Why do students take longer to conduct reviews?*

One reason novices take longer in reviewing may be that they are asked to compile, run and test their peers' solutions, taking more time than would be needed to simply read and evaluate a submission. Another reason may arise from students finding the work of their peers more valuable in novice programming than in other disciplines, and therefore spending more time observing the techniques and methods applied by their peers.

## 6.3 Future Work

In future semesters the findings of this evaluation will be used to improve the assignments and criteria used for peer review. Re-evaluation will be undertaken to measure any improvement.

The creators of PRAISE want to share the PRAISE system more widely with instructors. One possible avenue being pursued is to assist in improving the Moodle Workshop module which is languishing. Reinforcing the value of reviews by assessing the quality of reviews provided by students is another aspect of future development and investigation.

## 7 References

- Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., et al. (Eds.). (2001): *A Taxonomy for Learning, Teaching and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*. New York, USA: Addison Wesley Longman, Inc.
- Brook, C., & Oliver, R. (2003): Online learning communities: Investigating a design framework. *Australian Journal of Educational Technology*, **19**(2):139 - 160.
- The White Paper: A Description of CPR, Chapman, O. L. [http://cpr.molsci.ucla.edu/cpr/resources/documents/misc/CP\\_R\\_White\\_Paper.pdf](http://cpr.molsci.ucla.edu/cpr/resources/documents/misc/CP_R_White_Paper.pdf) Accessed February 23 2006.
- Chinn, D. (Year): Peer assessment in the algorithms course. *Proc. Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiSCE2005)* 69 - 73.
- Davies, R., & Berrow, T. (1998): An evaluation of the use of computer supported peer review for developing higher level skills. *Computers Educ.*, **30**(1/2):111 - 115.
- Nomination Statement - Awards for Programs that Enhance Learning, de Raadt, M., Loch, B., & Addie, R. <http://www.sci.usq.edu.au/staff/deraadt/award/>. Accessed 13th September 2007.
- de Raadt, M., Toleman, M., & Watson, R. (Year): Electronic peer review: A large cohort teaching themselves? *Proc. Proceedings of the 22nd Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE'05)*, Brisbane 159 - 168, QUT, Brisbane.
- de Raadt, M., Toleman, M., & Watson, R. (2006): An Effective System for Electronic Peer Review. *International Journal of Business and Management Education*, **13**(9):48 - 62.
- Dekeyser, S., de Raadt, M., & Lee, T. Y. (Year): Computer Assisted Assessment of SQL Query Skills. *Proc. Eighteenth Australasian Database Conference (ADC 2007)*, Ballarat, Australia 63:53 - 62, ACS.
- Hamer, J., Kell, C., & Spence, F. (Year): Peer assessment using aropä. *Proc. Proceedings of the ninth Australasian conference on Computing education (ACE2007)*, Ballarat, Victoria, Australia 43 - 54, Australian Computer Society, Inc.
- Kurhila, J., Miettinen, M., Nokelainen, P., Floreen, P., & Tirri, H. (Year): Peer-to-Peer Learning with Open-Ended Writable Web. *Proc. Proceedings of the 8th annual conference on Innovation and technology in computer science education (ITiCSE '03)*, Thessaloniki, Greece 173 - 178, ACM Press.
- Lewis, S., & Davies, P. (Year): Automated peer-assisted assessment of programming skills. *Proc. Second International Conference on Information Technology: Research and Education (ITRE 2004)* 84 - 86.
- Prins, F. J., Sluijsmans, D. M. A., Kirschner, P. A., & Strijbos, J.-W. (2005): Formative peer assessment in a CSCL environment: a case study. *Assessment & Evaluation in Higher Education*, **30**(4):417 - 444.
- Saunders, D. (1992): Peer tutoring in higher education. *Studies in Higher Education*, **17**(2):211 - 218.
- Sitthiworachart, J., & Joy, M. (Year): Effective peer assessment for learning computer programming. *Proc. Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education (ITiSCE2004)*, Leeds, United Kingdom 122 - 126.

# Computer Science in Context – Pathways to Computer Science

**Maria Knobelsdorf**

Institute of Computer Science  
Freie Universität Berlin  
Takustr. 9, D-14195 Berlin, Germany  
knobelsd@mi.fu-berlin.de

**Carsten Schulte**

Institute of Computer Science  
Freie Universität Berlin  
Takustr. 9, D-14195 Berlin, Germany  
schulte@mi.fu-berlin.de

## Abstract

In order to increase participation in Computer Science (CS), contextual approaches are often suggested for teaching. Although these approaches are quite promising, we do not know what exactly *context* means and how CS teaching should implement these approaches. In the broadest sense *CS in context* means that CS is linked to subject areas outside CS, helping students to perceive CS as a meaningful, useful, and helpful subject that is established in outside arenas.

The study we present in this paper explores the characteristics of *CS in context* that form possible pathways into the field. For this purpose, we analyse the computing experiences of students majoring in CS-related fields. The study is part of our research project about computing processes. In this project, we investigate students' computing experiences in order to understand how students' interests, motivation, and requirements for computing develop and how computing influences their understanding of CS.

In the current study, we examine general qualitative aspects of CS in context, especially activities and habits that sharpen and stabilize students' self-image and world-image. Because we find surprisingly few examples of specific contexts (such as subject areas) that are related to students' subject choice, we finish this paper with a discussion about possible reasons and conclusions for further studies.

**Keywords:** CS, Context, Pathway, Wider Access, Gender, Computers and Society, CS Ed Research, Pedagogy, Computer Biographies.

## 1 Introduction

Although considerable efforts have already been made to improve the situation, we are still encountering the same substantial problems in Computer Science (CS): decreasing numbers of beginners, constant high dropout rates, and a very low number of female students. In order to understand the reasons for this situation, a recent study of ours investi-

gated students' computing<sup>1</sup> experiences (Schulte and Knobelsdorf 2007). In this study, biographical computing processes of CS majors were compared with those of students not affiliated with CS. We found that CS-affiliated students align their computing experiences with CS, whereas unaffiliated students exclude CS from their computing experiences. This produces a world-image of computing and CS wherein affiliated students perceive themselves as insiders, whereas unaffiliated students perceive themselves as outsiders. Consequently, computing is a starting point for CS-affiliated students, but also a barrier for CS-non-affiliated students to take up CS studies.

Our last study revealed an interesting aspect: students frequently experience computing unhampered by any kinds of regulations, spontaneously, and outside formal schooling. This free leisure time environment represents a context where students experience computers and CS implicitly. For some students this context becomes a starting point to CS, while for other students it is a barrier. We were interested in this aspect of context and examined it in further detail.

Contextual approaches are often suggested in order to increase participation in CS. Fisher and Margolis conclude from their work that "the context of computing is often very important for women students. Among our sample, more women than men link their interest in computer science to other arenas such as medicine, the arts, space exploration, etc." (Fisher and Margolis 2002, p. 80). We can even exaggerate the argument in the opposite direction: CS majors often seem to have a narrower view of CS, they explicitly neglect the role of context, and they set CS apart from other subjects. However, such a narrow view of CS is often the most important reason for low participation in CS. Rosser argues that "female students will be more attracted to science and its methods when they perceive its usefulness in other disciplines" (Rosser 1990, p. 64). Beck, Buckner and Nikolova suggest that "[s]tudents taking CS courses do not wish to study computers as an end in themselves, but rather to become proficient in their use to the extent that they can use the computer as a tool to accomplish some other, non-computer-related goal" (Beck, Buckner and Nikolova 2007, p. 358).

Contextual-based learning is an established approach in science education. In his article *On the Nature of "Context"*

---

Copyright © 2008, Australian Computer Society, Inc. This paper appeared at 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. Raymond Lister and Simon, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

---

<sup>1</sup> With the term computing we refer to all kinds of computer usage and interaction with the computer.

in *Chemical Education*, Gilbert (2006) identifies a number of inter-related problems that chemical education has faced: content *overload*, learning of *isolated facts*, lack of *transfer* and *relevance*, and *inadequate emphasis* of the curriculum. “[T]he educational model that embodies the meaning of ‘context’ must be such that it provides an effective answer to the associated curricula and social problems” (Gilbert 2006, p. 958). Due to the fast accumulation of new scientific knowledge, we have to teach many concepts. This content load leads to teaching of isolated facts without supporting students to understand the correlation and meaning between the facts. As a consequence, we are faced with a lack of transfer and relevance. Students often do not know why they should learn the subject matter. Contextual-based approaches address these problems.

Although contextual-based approaches are quite promising, we do not know exactly what context means and how CS teaching should implement this approach. In the broadest sense, *CS in context* means that CS is linked to subject areas (application domains) outside CS and that it helps students to perceive CS as a meaningful, useful, and helpful subject that is established in outside arenas. But students’ motivation and interests depend on students’ prior experiences. In our research, we investigate students’ computing experiences in order to understand how their interests and motivation for CS were developed.

In the study presented in this paper, we are analysing computing experiences of students majoring in CS-related fields. We expect to find out more about *CS in context* as well as the role of context in providing pathways to CS.

Regarding the content, the paper is organised in three parts:

1. In section 2, we examine related work and discuss what *CS in context* means to CS Education.
2. Based on this discussion, we describe in sections 3 and 4 the research framework on which our study is based. This includes the theoretical background, our research instrument, the analysis procedure, and the participants of our study.
3. Finally, we present the results of the study in section 5.

The paper concludes with section 6 where we discuss the results and open problems.

## 2 CS in Context

Research on broadening participation and interest in CS is often done from a gender perspective (Camp 2002, Cohoon and Aspray 2006, Margolis and Fisher 2002). This work points out the idea of pathways and the importance of context. In the following paragraphs, we examine this aspect in further detail. For further reading about participation and interest in CS, see for example Carter (2006), Peckham, Harlow, Stuart, Silver, Mederer, and Stephenson (2007), Turner and Turner (2005), and Vegso (2005).

Usefulness seems to be a reason why CS should be linked to other disciplines. For many students, particularly female students, the usefulness of CS is not self-evident. Studies in

this field conclude that CS becomes useful when it is linked to other arenas, disciplines, or fields to accomplish non-computer-related goals. Rosser suggests “[using] methods from a variety of fields or interdisciplinary approaches to problem-solving” (Rosser 1990, p. 64). Fisher and Margolis argue too that “[s]ome of the elements of a more contextual approach include early experiences that situate the technology in realistic settings; curricula that exploit the connections between computer science and other disciplines; [...]” (Fisher and Margolis 2002, p. 81). The recommendations for CS teaching correspond to students’ observed requirements: teach CS with interdisciplinary contextual approaches in realistic settings. From a more theoretical perspective of CS Education we ask: what exactly does a contextual approach mean? Lave and Wenger (1991) addressed this question and developed a theory about situated learning, which we consider in the next paragraph.

Situated learning, as suggested by Lave and Wenger (1991), means that learning takes place within the community where the knowledge is used, as opposed to learning in conventional schools that “is predicated on claims that knowledge can be Decontextualized [...]” (Lave and Wenger 1991, p. 40). “In summary, rather than learning by replicating the performances of others or by acquiring knowledge transmitted in instruction, we suggest that learning occurs through centripetal participation in the learning curriculum of the ambient community. Because the place of knowledge is within a community of practice, questions of learning must be addressed within the development cycles of that community [...]” (Lave and Wenger 1991, p. 100). Learning is a process that starts with *legitimate peripheral participation* (LPP) and becomes central in the *community of practice* (CoP). Lave and Wenger developed this theory studying and observing traditional apprenticeship. They claim that LPP can be generally applied to learning. But they do not apply their theory to conventional teaching in schools; they just mention that it is decontextualized.

Guzdial and Tew (2006) analysed Lave and Wenger’s theory about situated learning in order to apply it to CS Education and teaching. They claim that the lack of legitimacy is probably the biggest problem of traditional school teaching. They argue that “[t]he best that we in traditional school can do is to *align* our instruction with students’ perceived community of practice [...]” Lave and Wenger’s theory suggests that students must perceive *some* alignment for learning to occur”. They continue that “[u]sing LPP as our theoretical perspective, we might ask what communities of practice do our majors perceive [...]” (Guzdial and Tew 2006, p. 52). Guzdial and Tew relate two substantial points of alignment: learning activities must be aligned with an external CoP and with students’ purpose and expectations. They argue that to incorporate the notion of a CoP into teaching is not sufficient; the incorporation must be meaningful and realistic for the students. From this argument we can deduce that learning *CS in context* means an alignment between learning and an external CoP, as well as students’ expectations, and this alignment must provide students with a sense-making perspective on the subject matter.

An alignment between learning and an external CoP can be understood as learning in realistic situations with realistic problems. However, beginners' notions of realistic CoP might be wrong or even opposed to CS. Moreover, in a fast-changing field such as CS it is difficult to provide students with realistic situations or problems. Ben-Ari (2004) argues that "what is a real situation will depend decisively on the students' background and future plans, amplifying the tendency towards premature determination of an occupation. It is also clear that it is impossible to present young students with situations that are really 'real' [...]. Especially in a rapidly changing field like CS, the specific content of secondary and even undergraduate education can become rapidly outdated" (Ben-Ari 2004, p. 88).

Ben-Ari points out that the content is not essential. A contextual approach is more than a compelling example, more than demonstrating the use of learning material in an application domain. It is more important to consider the roles people take on in a CoP: "The most important lesson that I draw from analysing situated learning in the context of CSE [Computer Science Education] is the importance of domain knowledge in most of the CoPs that students are likely to join [...]. Curriculum design should be more cognizant of what Shaw calls roles as opposed to content [...]. Situated learning supports her claim that students should choose a specialization that is oriented either to an application area or to CS technical expertise or looking to future managerial responsibility" (Ben-Ari 2004, p. 95).


But an effective or useful specialization depends on the students' perspectives: their backgrounds, their future plans, and their prospective roles in a CoP. Contextual approaches claim to provide effective pathways to CS because learning is aligned with students' expectations, their background, their development, and their biography. Therefore, likely more than one pathway exists. Fisher and Margolis (2002) argue that "[w]e need to establish the sense that there are multiple valid ways to 'be in' computer science" (Fisher and Margolis 2002, p. 81). Furthermore, while one pathway is meaningful for some students, it might be meaningless for others.

## 2.1 Summary

We summarize the current discussion about contextual approaches in CS Education. Table 1 shows the problem that contextual approaches refer to. The transition between the different phases is very difficult. The initial phase, where interest and motivation for a subject or area grow, is highly contextualized. During high school, college, and university, students learn subject matter decontextualized. Finally, when they finish their studies, they are faced with a highly contextualized employment or job. CS learning material should be linked to a context in order to provide a better transition from one learning phase to another.

A contextual approach links learning material to activities in realistic settings and CoPs. This approach enhances the quality of teaching and learning because it demonstrates the usefulness of the learning material, and this motivates students. However, until now only vague guidelines have been

Initial phase	High school/ University	Employment/ Job world
Highly contextualized	Decontextualized	Highly contextualized



**Table 1:** Connection between the different learning phases in life and the way learning material is studied and taught

developed for implementing contextual approaches. Therefore the question remains open what kind of context should be chosen in order to effectively implement a contextual approach. Given the current discussion in CS Education, several perspectives are possible:

- **Hypothesis 1.** *A context is effective if it aligns learning material with students' current interests.* The alignment between context and students' interests means that a context must be found that shows students how to use CS in areas the students are currently engaged in. For example, if students are engaged in biology, CS applications and activities in biology will be very effective. If they are not interested in biology, a context from biology will be useless.
- **Hypothesis 2.** *A context is effective if it aligns learning material with examples students can accept as realistic.* Students need to understand how CS is used in practice. Therefore, a context is effective if it provides examples which students accept as realistic and meaningful. For example, if students are able to accept that CS is used in biology, biology will be an effective context, regardless of whether the students are interested in biology.
- **Hypothesis 3.** *A context is effective if it aligns learning material with students' current interest in a CoP.* Depending on the roles members play in their CoP different ways of using CS in an application domain are possible. A context is effective if it supports students in exploring roles they want to play. For example, if students are engaged in biology, a demonstration of the different roles and activities a biologist plays and takes on will be very effective.
- **Hypothesis 4.** *A context is effective if it aligns learning material with realistic examples of possible CoP.* Possible roles are demonstrated to students. Students are introduced to a possible CoP where they can become familiar with different roles and activities in authentic situations. For example, if students can perceive possible future roles in biology, biology will be an effective context, regardless of whether students are currently interested in becoming biologists.

In summary, the question is whether teachers should adjust subject matter to students' interests in other disciplines, or only to realistic problems or settings. To put it differently, learning to solve their prevailing problems by using CS can motivate students, as can learning to solve the problems of 'others' in this way. The other question is whether students

are more interested in contextualized learning materials or in the possible roles, responsibilities and activities such materials provide.

In the following section, we introduce very briefly the research framework that forms the context for the study of this paper. Thereafter, section 4 contains a description of our study's participants and research questions. Finally, in section 5, we present the results of the study with regard to the four hypotheses.

### 3 Research Framework

This study is part of a larger research project that assumes that today's students – whether in K-12 or at university level – enter the CS classroom with preconceptions of CS. The research project investigates students' everyday contexts that provide or influence students' conceptualizations of CS.

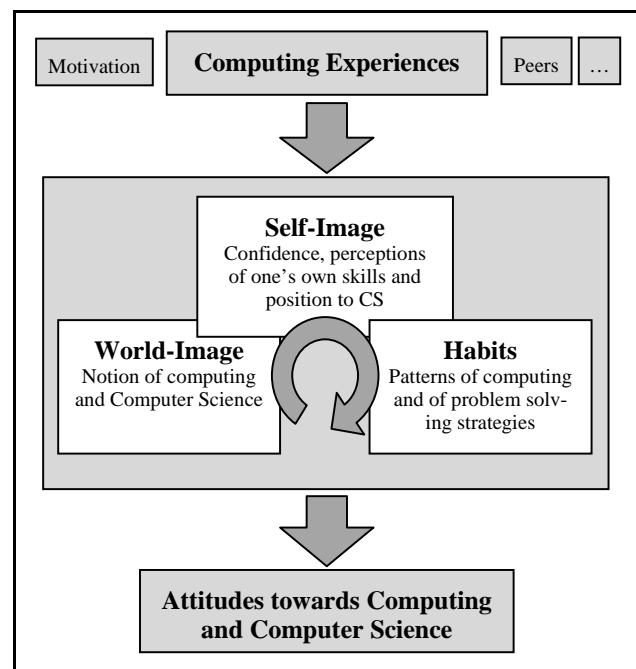
Novice conceptualizations often differ from the scientific concepts. In science teaching, theories of novice conceptualization processes and conceptual change play a major role in understanding and supporting learning. Conceptual change is generally defined as a form of learning that changes an existing understanding. Students already have an understanding (conception, belief, idea, or way of thinking) about the subject, which in educational research is called pre- or misconception. Teaching under terms of conceptual change primarily implies uncovering student preconceptions in order to help students to change their conceptual framework. Conceptual change theories seem to be useful in demonstrating the importance of taking into account students' prior knowledge and incorporating students' everyday contexts into teaching.

Under the terms of conceptual change, learning is not only a cognitive acquisition of knowledge. It also includes and affects all aspects of a student's personality: the student's personal story, self-perception, and view of the world, as well as habits, and learning styles. We assume that conceptions of and beliefs in CS are developed through a learning process which, among other factors, is influenced through computing in formal and informal settings. In our research project at the *Institute of Computer Science, Freie Universität Berlin*, we aim to understand the conceptualization process of CS in order to develop didactical interventions for CS teaching. We want to understand the whole process of computing and its role and impact on students' learning processes in CS. To this end, we have developed a biographical research approach where we survey students' personal computing stories. Our data gathering method provides autobiographical computing narrations in written form, which we call *computer biographies* (Computer-Biographies, Webpage). In the following section, we describe this biographical perspective and the way it is connected to the *contextual approach*.

#### 3.1 Biography as a Method

A computer biography is a story a person tells about his or her computing experiences. Typically, a story is told from a personal, subjective point of view and contains only those

aspects that the author considers to be valuable and important for the story. In particular, when we ask CS majors to write down their own computer biography, they are implicitly triggered to write about those experiences that explain why and how they became CS majors. Such texts usually follow a typical narrative pattern: starting with a beginning such as the first contact with a computer and ending with the current situation, for example, with a happy ending. In between we find important experiences that fostered or constrained their development. We also find information about computing experiences and activities. As computing and CS are closely related (especially for novices), computer biographies reveal information about conceptualisations of CS.



**Figure 1:** The analytical categories self-image, world-image, and habits as specifications of the biographical computing process (Schulte and Knobelsdorf 2007, p. 32)

From a more scientific point of view, computer biographies form a qualitative biographical research design to explore computing experiences and their influence on people's belief systems related to CS. Such a belief system is made up of a person's self-image, world-image, and habits (see Figure 1). The *self-image* includes self-conception and judgement, as well as attitudes regarding the subject's own computer skills and orientation in the computer world. The *world-image* comprehends personal theories and preconceptions about computing and CS. *Habits* comprehend learning strategies, typical performances with the computer and reactions to problems (Schulte and Knobelsdorf 2007, pp. 31).

In the next section, we present our empirical study, which connects the theoretical framework about *context* with the biographical research framework.

## 4 Current study

In the current study presented in this paper, we try to apply the analytical categories world-image, self-image and habits to explore the relevance of context in computer biographies. Referring to the discussion above, a context might be effective if learning material is related to useful applications that students can experience by using the computer. Consequently, analysing students' computing activities should reveal the relevance of contexts. It should also reveal the relevance of roles in a CoP, as roles are determined by the typical activities of a member of a CoP.

We chose a group of students who had just recently enrolled in a CS-related subject in order to focus on students who are dependent on context more than others. We expect bioinformatics students to talk about computing experiences related to biology, bio-technology, or biochemistry; we expect them to want to locate computing in the context of biology. Likewise, we expect mathematics majors to talk about theoretical, logical, and abstract issues concerning CS. Finally, we expect CS education majors to talk about the reasons why they chose to become future CS teachers.

### 4.1 Participants

In order to investigate contextual approaches to CS, in our study we examine computer biographies of students majoring in CS-related fields: bioinformatics majors entering university (25 men, 22 women), mathematics majors with CS as a minor subject (5 men, 5 women) and second-year CS Education majors (19 men, 3 women). The fact that these students have such varied interests makes them a highly interesting study object for our research purpose.

"Bioinformatics is the application of computational tools and computer technologies to model, analyse, store, retrieve, manage, present, and visualize biological data" (Zhang, Lin, Olsen, and Beck 2007, p. 186). Bioinformatics covers biology, bio-chemistry, and CS, and refers to hypotheses 1 and 3 concerning an effective context. Therefore bioinformatics is ideal for students who are interested in all three subjects.

German mathematics majors are obliged to take a minor subject, which very often is CS or physics. CS and mathematics are strongly related to each other, and there is anecdotal evidence that many mathematicians perceive CS as applied mathematics.

In Germany, the subject *CS Education* focuses on teaching CS in high school. Every teacher is obliged to study CS, education, and another major subject such as mathematics, history, sports, or sociology. We hypothesize that these student groups are even more likely to enjoy contextualized teaching approaches than other students because their subject choice shows that they wish to align different areas of their interest. Therefore they should be highly interested in learning how to apply CS in other areas (contexts).

First step	Extraction of basic information needed to build a coding system
Process of data access	Gaining overview about roles and contexts in biographies Finding important topics and structures for first potential codes
First coding	Material coding based on the considerations above
Second step	Material coding
Setting up code system	Development of explicit codes and coding rules based on first coding (activities and periods)
Second coding	Two independent coders coded all material, applying the developed coding rules and codes
Third step	Interpretation of coded results
Relating codes	Relating activities and periods and their interpretation concerning roles and context
In-depth interpretation	Focusing on important activities in decision period

**Table 2:** Analysis and interpretation procedure, the coding steps and their illustration

### 4.2 Analysis Procedure

As described in section 3.1, we connect the theoretical framework for context with our biographical research framework. In order to operationalize context, we investigate activities. Computing activities means uses of the computer in a certain context. When we investigate computing activities we should find out something about students' perspectives on context. Students not only describe their computing activities but also give feedback on how they enjoy them. These personal opinions about students' experiences can express a certain position that refers to a CoP (Community of Practice). A context is also effective if learning material is aligned to a CoP. This alignment is specified in the activities carried out by members of a CoP and in the roles these activities are related to. Furthermore, activities and habits determine roles. Therefore, in the analysis process we focus on activities and related roles.

The analysis was divided into three consecutive steps. In the first step, we read the material in order to get a general

idea of relevant information and to identify potential codes for the coding system. Here, we relied on concepts of coding systems from prior biographical studies. During this process, we observed that the biographies partly resemble each other in their structures.

In the second step, we set up and refined the coding system, and testing it until we had a final result: we coded the material according to the activities described and for three different parts or periods of a computer biography.

In the third step, we examined all activities in their respective periods for frequency, particularities, or length. Thereafter, analysis was narrowed down to focus on more specific aspects of the biographies: the relevance and characteristics of the most important activities with regard to role and context. In Table 2 we summarize these steps.

The results of this analysis and interpretation are presented in the next section.

## 5 Results

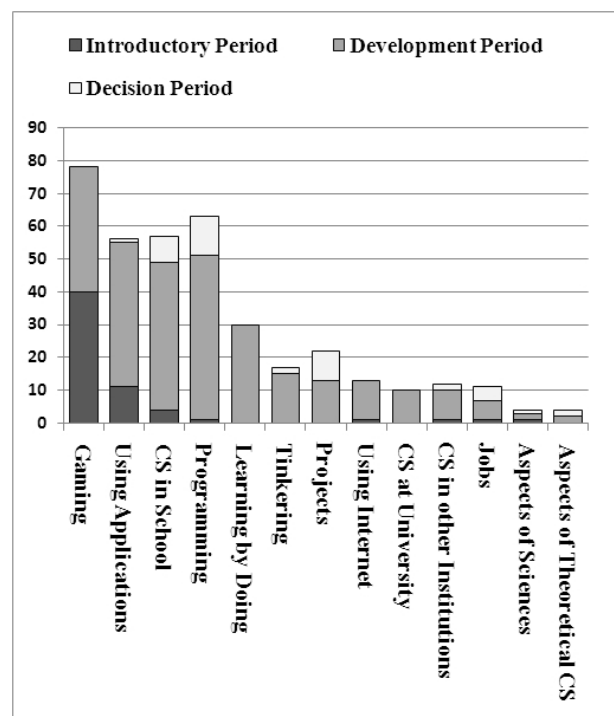
In this section, the results are presented in a way that roughly reflects the analysis procedure. First, we present an overview of activities analysed in the first coding step (section 5.1). Then we describe in section 5.2 how we divide the biographies into different periods. In this second analysis step, we reveal only some hints referring to context and roles. Thereafter in section 5.3, the analysis is narrowed to examine the most important period in more detail.

### 5.1 Results of the First Analysis Step

In 47 biographies of bioinformatics students, we found only four students mentioning aspects of science. Two of them relate computing to biology; the other two combine two different fields of interest (biology and computing). In the other 43 biographies biology or science are not mentioned.

Similar observations can be made in the group of math students. One student mentions his interest in theoretical computer science. We interpret this as an indication of a mathematical context. Two students regard CS as important in order to cope with the course *computational mathematics*. We can also interpret this as an allusion to CS (or computing) in context; probably the interpretation ‘math in context’ would be more accurate here. Overall, three of ten biographies contain indications of a context; however, these indications are rather vague.

Some CS Education majors mention that they found CS at school interesting, but only three students link their computing experiences to their subject choice. One student considers his CS teaching in school as didactically valuable and therefore wants to study CS Education himself. Another biography claims the opposite: because of negative impressions of the teaching quality in school, the student wants to become a CS teacher and do a better job. The third biography claims that apparently CS teachers are not obliged to program so much. In addition, several students mention that they enjoyed their CS courses at school without explicitly linking these experiences to their subject



**Figure 2:** Activities and Periods (the y-axis gives the total nomination number of each activity in all biographies surveyed)

choice. We find some explicit references to subject choice, but not as many as expected.

### 5.2 Activities and Periods

The biographies are structured like a narration with a beginning, a climax, and a (happy) ending. Our analysis of these patterns, together with related activities, reveals three periods in the biography. We call the first part the *introductory period*. This period starts with the first computer contact. It contains experiences and situations that are initiated by accident or by others. After the introductory period, a *period of development* begins. This period is characterized by purposeful experiences where students develop their interests. During this development a *decision period* might take place. A decision period contains important experiences that are decisive for the future. Such experiences are described in detail and are much longer than other experiences and events in the biography.

For all periods we examine the activities and experiences the students have had with the computer. We also examine how the students perceive their own activities and experiences. This helps us to identify which type of activity was experienced in which period and had most influence on the students. Figure 2 shows the types of activity according to the different periods.

#### 5.2.1 Introductory Period

The introductory period starts with the first computer contact and ends when a certain development, which does not

happen accidentally, can be observed. The biographies typically start with a description of the first encounter with computer, the surrounding events, and the experiences of the student. The introductory period contains experiences and situations that are initiated by accident or by others. The transition to the next period, the period of development, is triggered mostly by the fascination for computers and the challenge they offer to create something by oneself. This transition is mostly smooth. Sometimes the change is indicated by a new paragraph or keywords that mark the beginning of something new in the story.

In 77 biographies we found 63 introductory periods. The 14 biographies without this period were either very short (1-2 sentences) or started directly with the period of development. The students typically first used a computer between the ages of eight and eleven. Almost always the first computing activity is gaming. Most students mention gaming (40 times) as their major activity in this period, considering it as interesting (9) and fun (7). The activity most frequently mentioned after gaming is the use of applications (11). Some students experience the first computer activities at school (4), such as the use of applications. But the majority encounter the computer outside school, mostly at home. Parents, brothers and sisters, other family members, or peers help them and explain the computer usage. Usually this *supportive* person is male. Very often students mention family members' professions when they belong to computer science or engineering, probably in order to explain why they themselves are interested in CS, too.

Our overall impression of the introductory period is that all of them are very similar and also very short in comparison to the time period they describe. The central activity in this period is gaming. After the first coding process we observed that the introduction to computing happens mainly by chance. Therefore, we define the introductory period as being accidental. Indeed, in nearly all biographies the first contact was accidental. Only in some biographies can a kind of purposeful introduction to computing be detected; these biographies were coded as beginning directly with the development period. With regard to roles and context in this introductory period we summarise:

<b>Dominant context</b>	leisure time
<b>Typical activity</b>	using applications (games)
<b>Typical role</b>	beginner, learning, passive, introductory help by others, typically male person

### 5.2.2 Period of Development

After the introductory period, we typically find a period that is marked by a certain development. The period of development is characterized by purposeful experiences. Students guide their development instead of being guided by others or by accident as in the introductory period. In 77 biographies we found 81 periods of development, because in four biographies this period was interrupted by the deci-

sion period and therefore contained two periods of development.

In the period of development many different activities are described. Sometimes they are just enumerated. Almost all school experiences and activities take place in this period. The programming activity occurs in this period. These are the activities that were mentioned the most: programming (50 times), CS in school (44), using applications (43), gaming (37), and tinkering – especially with hardware (15).

Many students describe a great fascination for computing and a great interest in exploring and learning more about computers. Interests, satisfaction, or fascinations were mentioned 67 times in this period. For a better illustration we quote text examples from three biographies:

*"[...] my interest grew again because of the seemingly never ending possibilities with this tool."* [101B1986m]<sup>2</sup>

*"The possibilities the computer offers are uncountable."* [92B1988m]

*"At this time I was inspired by the creative possibilities of computers."* [87B1985m]

But this period also contains ups and downs in which students lose their interest in CS for a while or are bored by it. We found text samples in which students described disappointing (16), boring (8), and uninteresting situations (6) without fun (2). Problems, however, are perceived as challenges to learn and to explore more. *Learning by doing* activities were mentioned in 30 biographies, but 27 times this was in the period of development. Learning is perceived as widening the possibilities and skills and therefore rewarding. New experiences and activities trigger students to learn more and to gain more new experiences. Furthermore, the students describe self-confident and independent habits regarding solving computer problems and learning programming. They also appear to be confident (especially women) of coping with a major in the subject.

These learning habits are common to most CS majors we have investigated so far in previous studies. But unlike many CS majors, the students majoring in CS-related fields do not divide computing into *using* and *designing*, in which the latter is superior and reserved for computer scientists; the computing experiences seem to have less impact on their self-image. In comparison, it seems that computing experiences and self-perceived computing skills are very closely connected to the self-image of CS majors; their identity seems to rely quite heavily on the self-image in computing, whereas it does not for the students majoring in CS-related fields.

Our overall impression of the period of development is that it starts with activities students are interested in. They re-

<sup>2</sup> This code identifies a biography. The first number is the biography number; the upper-case letters refer to students' major subject and are B for bioinformatics, CSE for CS Education, or M for maths; the second number is the student's year of birth; the lower-case letter is f for female or m for male.

gard these activities as an opportunity to explore the computer's possibilities and to learn more about it, for example, programming as a starting point. Students show an addiction to exploring the computer, learning by doing. This improves their self-image and provides a positive world-image (computing is fun, creative, self-exploring). The students start as users and become designers without seeing a dichotomy between these two aspects of computing. Becoming a designer is not a change from one group to another but a development from simple to complex activities. With regard to roles and context in the period of development we summarise:

<b>Other contexts</b>	school, learning, homework
<b>Typical activity</b>	more applications, also programming as a new computing activity
<b>Typical role</b>	roles are changing: advanced learner, more active, problem-solving and exploring

### 5.2.3 Decision Period

The decision period is typically characterized by a special experience that is decisive for the future. This experience is described in more detail than other experiences and events in the biography. We determined that only one decision period can take place. This period can be detected by looking for formulations containing superlatives, differentiations, and keywords such as 'everything changes', 'special', 'other', 'new', 'important'... A new paragraph can also indicate a break or change in life. Experiences that are described in detail or differ from the chronological description of the rest of the story are also evidence for the decision period. The decision period is characterized by a high degree of students' self-determination. Ups and downs are no more mentioned. Instead, the students restrict themselves to aspects that are important or 'decisive' for their biography. There was no decision period that was not preceded by a period of development. The decision period is like a consolidating climax of the period of development. Therefore we should rather have named this period the *period of consolidation or summarization*.

In the period of development many different activities were described. In the decision period the students concentrate on a few important activities such as programming or project activities. Internet, tinkering, and games are not mentioned any more, or are no longer seen as important. In the 20 decision periods we analysed, the following activities are mentioned most often: programming (12), CS in school (8), doing projects (8), and doing a student job (4). These activities are rated as being interesting (9), fun (6), and satisfactory (4).

Altogether, we found 20 decision periods in 77 biographies. We observed that the biographies without a decision period very often end with a single paragraph or sentence in which the students describe a decision. In this final part the students explain why they decided to major in their subject:

*"Because computer science was too dry for me and I was also interested in biology and chemistry, I chose to major in bioinformatics."* [109B1986m]

*"For a long time I wasn't interested in computer science, but during my PhD in medicine I realized that in our research group projects were more successful when they were done with computers."* [107B1971m]

*"I became curious and decided to major in this subject."* [140M1986f]

But these text samples are not (!) decision periods. The decision period is a biographically established decision, not an explanation of subject choice. The decision period is characterized by a consolidation of interest in CS, and we assume that this decision is permanent or consistent for a while. With regard to roles and context in the decision period we summarise:

<b>Other contexts</b>	school, projects, jobs
<b>Typical activity</b>	focusing on a single computing activity
<b>Typical role</b>	role changes toward expert: competent usage, perceived as satisfying

## 5.3 Interpretation of Activities in the Decision Period

As we have seen in section 5.2.3, in the decision period one type of activity often becomes central, and is a major factor for the decision to study CS. In this section, we explore these important activities in more detail, and interpret them with respect to context and related roles. The activities to be analysed are: programming, learning CS at school, and jobs and projects.

In a decision period the most prominent activity (the one most often mentioned and described) is programming. Programming is closely related to CS at school and to projects. CS courses at school can be considered as positive, although at the same time students claim to have had to learn programming on their own, or that the teacher was often not able to explain the subject matter. Therefore, the role of CS in high school is critical: while teachers are regarded as incompetent, they introduce many students to programming and are able to increase their interest, motivation, and programming skills; in many cases, programming influences the subject choice and is explicitly mentioned:

*"I did a lot of programming in my leisure time; it's the main reason for my decision to study [CS Education]"* [16CSE1981m]

At first sight, no links between programming and context or roles as well as statements referring to specific contexts can be found. Only one student is exceptional. He is a student of Bioinformatics, argues for the importance of free software, uses only such (like FreeBSD), is a member of the free software foundation Europe, and programs in C++ for

FreeBSD. These activities are an example of participation in a CoP. However, the link to Bioinformatics seems only marginal. Apparently, the aspect of fun is an important reason for the popularity of programming. Students enjoy to solve a task or problem, thereby creating a product or at least an artefact. The next text sample illustrates this:

*“Programming [in CS class at school] was always fun because in most cases a running piece of software was produced.” [14CSE1983m]*

So, programming positively influences students’ self-image in two ways. First, it is fun and students want to engage in computing and CS. Second, this experience demonstrates students that CS is something they are good at:

*“As my CS course at school needed a month to cover only if/else-constructs in Pascal, I knew it is my task to study it on my own. During fall holidays, I took a book of programming and studied learning material of the whole school year [in two weeks]. However, the CS course at school wasn’t useless, because the teacher helped me to improve my programming skills, and gave me more difficult programming tasks.” [135M1987m]*

Other biographies refer to programming contests, which gave students the opportunity to overcome self-consciousness, thus positively influencing their self-image. In several biographies students mention that it was rewarding to compete with other students during programming. Another example may illustrate this aspect:

*“My interest for programming rose when my CS course at school started. The whole course was about programming [...] At home I added additional functionality to the programs, which was beyond the required tasks.” [09CSE1985m]*

We analysed project activities because projects are probably closely connected to roles and contexts – at least projects are embedded in an application area. The application areas mentioned are: school homepages, building and configuring a network in the school building, robotics, and other application areas described only vaguely, such as web-related projects. The last cited biography above, for example, continues with a description of two interesting projects. One project dealt with the development of software for Kephera robots in cooperation with a local university. The other project was set up in cooperation with a local company where the students participated in an R&D project involving adaptable lights for cars. Such projects often start in CS courses at school. But we did not find projects that were confined to a CS course. There were examples of cooperation between school and other institutions, or between the CS course and another subject.

When biographies contain descriptions of projects, they refer to certain contexts, but these seem to be important only as triggers for programming activities such as robotics or an interactive homepage for the school. The functionality of CS at school, in projects, and in jobs seems to provide incentives to start programming.

Altogether we found only a few biographies mentioning projects in a CS course at school. This aspect is somewhat

confusing, as projects are of great importance in the German tradition of CS teaching at high school. For example, in Berlin, where many of the participants of the study supposedly attended high school, an entire school semester should be devoted to a project. Perhaps fewer projects are done in schools than are required by the curriculum, or these projects are too irrelevant for the students to include them in their biographies. Overall, projects and jobs are often described as motivating, fun and relevant, but the description often remains superficial, without details.

So far, we have discussed the significance of context as an application area. Another approach focuses more on roles than on context. Now we re-examine the above-mentioned activities from the perspective of roles.

We found only sparse allusions to roles in the biographies. For example:

*“I decided to study CS for teaching at high schools because I didn’t want to become a programmer.” [29CSE1983m]*

*“As both CS teachers were incredibly incompetent, I became motivated to show that it is possible to teach the subject matter in a reasonable way and to engage also those [pupils] who are marginally interested in [CS]” [10CSE1975m]*

The third allusion to roles was already mentioned above: programming as an activity to contribute to Free Software. All of these allusions are rather vague. Overall we did not find sufficient information about our hypotheses three and four. Instead, it seems as if none of the hypotheses could be confirmed; perhaps roles are not so important, after all. In the next section we discuss this aspect in further detail.

## 6 Discussion

In this study we wanted to find out more about the importance and characteristics of context and the student perspective on context. Based on related work, we supposed that contextual approaches were important and therefore somehow visible in computer biographies of students majoring in CS-related fields.

Referring to the four hypotheses from section 2.1, we have some interpretations based on the results of our study:

**Hypothesis 1.** *A context is effective if it aligns learning material with students’ current interests.*

**Hypothesis 2.** *A context is effective if it aligns learning material with examples the students can accept as realistic.*

**Hypothesis 3.** *A context is effective if it aligns learning material with students’ current interest in a CoP.*

**Hypothesis 4.** *A context is effective if it aligns learning material with realistic examples of possible CoP.*

Our results reveal that the students do not explicitly connect computing with usefulness. We have not found any substantial data showing that the participants link computing to a context – either with current interests or with realistic examples (hypotheses 1 and 2). Computing is not closely related to specific application areas; although the bioinform-

matics students obviously aim at career prospects that place CS in the context of biology, they do not explicitly ask to learn CS in a biological context. They are simply interested in the general possibilities offered by computing. It seems that the activity itself is enough motivation, and no additional incentives are needed.

With regard to the importance of context to providing pathways into the CS field, we found that rewarding and motivating activities are important. We cannot say whether, for example, gaming is important as a first motivating computing experience, but it is typical and somehow it seems to spark interest in exploring more related experiences. One of these can eventually be programming – in most cases, triggered by external factors, such as the opportunity to enrol in a CS course.

## 6.1 Discussion of Methods

Nevertheless, we found surprisingly little information about context and activities related to roles. Some possible reasons are listed below.

1. We asked the wrong questions (data gathering method was wrong).
2. We asked the wrong students (the population should be different).
3. We asked too early (data gathering should be later in the students' studies).
4. We asked too late (data gathering should be earlier, in high school).
5. We analysed data wrongly (data analysis procedure was wrong).
6. We did everything right, but contextual approaches are simply not so relevant for students.

The use of computer biographies certainly has some implications. This method is based upon texts in which participants describe their experiences with computers. These are individual memories of past activities at the computer. This empirical method claims that such texts are written narrations and that they contain a story line. Not only isolated facts and descriptions are given, but also additional information such as feelings or opinions. For example, we found comments such as:

*"In high school I chose to focus on biology as my main subject and became enthralled for the genetic code; to me, it is the perfect programming language."* [108B1985m]

While this text sample is a reference to programming, it is also an example of additional information included in a biography that extends the focus of describing computing. Such additional information is quite common.

References to context and roles are typical examples of such *additional information*. Having found hardly any information on contexts, we have to ask whether this issue is caused by the instrument used (see reason 1). Do computer biographies contain the additional information as described? The answer is: yes they do. This can be seen, for

example, in section 5.2, where additional information concerning activities is summarized.

While biographies contain additional information in general, perhaps they do not contain information about context and roles. Context is somehow different from other additional pieces of information like feelings, peers, problem-solving strategies, characterizations of the computer and of CS. We do not see such a qualitative difference between references to context/roles and references to other additional information in computer biographies. Some information as to the context is even given directly (see section 5.1). In summary, we do not think that the results are caused by an incorrect empirical approach.

Another reason for the result might be the population studied (see reason 2). Perhaps the students we asked to write their computer biographies were somehow not able to include information about context. We deliberately focused on students who were not majoring in CS but studying CS in context with another subject (teaching, math, and biology). The argument was that students who chose to study not just CS but 'CS plus something else' were even more likely to acknowledge contexts. Given the results, this might be wrong, but the reason is unclear.

Similarly, we possibly asked them at the wrong time (see reasons 3 and 4); if we had asked them earlier or later in their lives, the results might have been different. A biographical narration is written from the specific perspective of a person at the actual point in the present. This perspective organizes how the past is conceptualized. By changing this point of view, we expect the story told to be different. In fact, this aspect is the reason why computer biographies are not objective descriptions of facts, but instead contain a richness of implicit information (what we called *additional information* above). We surveyed students who had just entered university (math and bioinformatics group) and students who had already studied for several semesters (CS teacher group). It may be that freshmen indeed focus more on the subject they are about to engage in. Therefore they focus in their computer biographies on the issue CS, because we asked them from a CS perspective, on their first day in university, and during a CS course, to write their computer-biography.

We indeed have the impression that biographies from students who intend to become CS teachers contain more information about context. On the other hand, this might be due to the fact that the students have a more precise perspective of their future roles in their jobs as teachers. In conclusion, we admit that perhaps we would have found a different picture if we had asked this population at another point in their lives. Nevertheless, and this is the crucial point, this does not explain why, at this point, they wrote so little about context.

Another explanation could be that we did not use the right analysis procedure (see reason 5). We chose to analyse the described activities in the biographies in order to find out more about context. We concentrated on activities because biographies comprise mainly activities. Based on the related work in section 2, we concluded that activities would be the

most obvious aspect to look for in order to gain information about context. Perhaps another approach is possible and our operationalization of context must be revised.

## 6.2 Conclusion

In their biographies, the students write about their computing experiences. The stories focus on computing and not on context. This could be the reason why we did not find much information related to context. On the other hand, biographies are likely to include many additional thoughts and comments – whatever kind of information the author thinks of.

Our interpretation of the results is that students do not perceive roles and contexts as being important – otherwise we would have found more information about it (see reason 6 from section 6.1). We interpret the results of the study as a hint that even freshmen who enrol in CS-related fields do not link CS to contexts. In general, they simply do not know how CS is embedded in contexts; how, for example, CS is used to solve biological problems. We had the impression that students relate their computing activities with the context *computing*, and their computing activities were limited to the computing context. In addition, they were not able to relate their own computing experiences to contexts, and rather perceive computing as a kind of closed world, detached from its surroundings.

What is the consequence of this interpretation when it comes to teaching? Can we conclude that teaching in context is not important because context is presumably not a central factor in the awareness of students? On the contrary, if the interpretation of the results in the paragraph above is correct, then the students do not perceive CS *in context* although they study CS-related fields. A conclusion therefore is that students perhaps choose to study CS-related fields because their interest in computers, computing and CS was not enough to perceive a future Community of Practice that they could wish to belong to.

From our recent study we know that CS majors tend to have a fixation on computers. Perhaps the students majoring in CS-related fields tended to reproduce this belief system as well. Therefore they chose a CS-related subject because they were not identifying with this ‘fixation on computers’. But then this conclusion supports the belief that students have a very narrow perspective of CS; it is then even more important to use contextual approaches in teaching. Altogether, we conclude that more discussions and studies are necessary to explore and understand students’ perspectives on CS and the promising contextual approach in CS teaching.

## References

- Beck, J., Buckner, B. and Nikolova, O. (2007): Using Interdisciplinary Bioinformatics Undergraduate Research to Recruit and Retain Computer Science Students. *Proceedings of the 38th SIGCSE technical symposium on Computer science education: SIGSCE '07*, New York, USA: 358-361, ACM Press.
- Ben-Ari, M. (2004): Situated Learning in Computer Science Education. *Computer Science Education* **14**(4): 85-100.
- Camp, T. (2002): The Incredible Shrinking Pipeline. *ACM SIGCSE Bulletin* **34**(2): 129-134.
- Carter, L. (2006): Why Students with an Apparent Aptitude for Computer Science Don't Choose to Major in Computer Science. *Proceedings of the 37th SIGCSE technical symposium on Computer science education: SIGCSE '06*, New York, USA: 27-31, ACM Press.
- Cohoon, J.M. and Aspray, W. (2006): *Women and Information Technology: Research on Underrepresentation*. Cambridge, Mass., MIT Press.
- Computer-Biographies: Webpage of the research project. <https://www.inf.fu-berlin.de/w/DDI/ComputerBiographies>
- Fisher, A. and Margolis, J. (2002): Unlocking the Clubhouse: The Carnegie Mellon Experience. *ACM SIGCSE Bulletin* **34**(2): 79-83.
- Gilbert, J.K. (2006): On the Nature of “Context” in Chemical Education. *International Journal of Science Education* **28**(9): 957-976.
- Guzdial, M. and Tew, A.E. (2006): Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education. *Proceedings of the 2d International Computing Education Research Workshop: ICER '06*, New York, USA: 51-58, ACM Press.
- Knobelsdorf, M. and Schulte, C. (2005): Computer Biographies – A Biographical Research Perspective on Computer Usage and Attitudes towards Informatics. *Proceedings of the 5th Koli Calling Conference on Computer Science Education: Koli Calling '05*, Turku, Finland: 139-158, TUCS General Publication.
- Lave, J. and Wenger, E. (1991): *Situated learning: Legitimate peripheral participation*. Cambridge, Cambridge Univ. Press.
- Margolis, J. and Fisher, A. (2002): *Unlocking the Clubhouse: Women in Computing*. Cambridge, Mass., MIT Press.
- Peckham, J., Harlow, L.L., Stuart, D.A., Silver, B., Mederer, H. and Stephenson, P.D. (2007): Broadening Participation in Computing: Issues and Challenges. *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education: ITiCSE '07*, New York, USA: 9-13, ACM Press.
- Rosser, S. (1990): *Female Friendly Science: Applying Women's Studies Methods and Theories to Attract Students*. New York, USA, Pergamon Press.
- Schulte, C. and Knobelsdorf, M. (2007): Attitudes towards Computer Science-Computing Experiences as a Starting Point and Barrier to Computer Science. *Proceedings of the 3rd International Computing Education Research Workshop: ICER '07*, New York, USA: 27-38, ACM Press.

- Turner, E.H. and Turner, R.M. (2005): Teaching entering students to think like computer scientists. *Proceedings of the 36th SIGCSE technical symposium on Computer science education: SIGCSE '05*, New York, USA: 307-311, ACM Press.
- Vegso, J. (2005): Interest in CS as a Major Drops Among Incoming Freshmen. *Computing Research News* **17**(3): 126-140.
- Zhang, M., Lin, C.-C., Olsen, G. and Beck, B. (2007): A Bioinformatics Track with Outreach Components. *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education: ITiCSE '07*, New York, USA: 186-190, ACM Press.

# Students' understandings of concurrent programming

Jan Lönnberg<sup>1</sup>

Anders Berglund<sup>2,1\*</sup>

<sup>1</sup> Department of Computer Science and Engineering  
Helsinki University of Technology,  
Espoo, Finland,  
Email: jlonnber@cs.hut.fi

<sup>2</sup> Department of Information Technology  
Uppsala Computing Education Research Group, UpCERG  
Uppsala University,  
Uppsala, Sweden  
Email: anders.berglund@it.uu.se

\* Temporary affiliation

## Abstract

This paper describes a qualitative, explorative study of how students understand some concepts in concurrent programming. The study is based on interviews with students regarding the final programming assignment in a concurrent programming course. We use phenomenography to analyse the students' statements about tuple spaces, the concurrent data structures on which the assignments are based, and to find the different ways in which they understand writing and debugging a concurrent program. We then discuss the effects of these understandings on how students construct concurrent programs, how teaching can be improved to form more useful understandings and how software tools can be designed to support the development of concurrent programs.

## 1 Introduction

Concurrent programming is both an important technique and a challenge. On the one hand, concurrent programming provides a way to make effective use of parallel and distributed systems and structure systems that perform many simultaneous tasks. On the other, the unpredictability of interaction between concurrently executing processes also introduces many pitfalls in the software development process that may result in software defects that are hard to find.

The research presented in this paper is part of a larger project with the long-range goal of making it easier to produce correct concurrent programs (i.e. programs that consistently produce the right results despite the aforementioned unpredictability) by supporting the detection and elimination of defects (or *bugs*).

In order to effectively develop methods and software tools to help find and eliminate bugs in concurrent programs, it is necessary to understand the bugs that can appear in these programs and the programmers' reasoning underlying these bugs. While insights about bugs per se can be gained by examining code, an understanding of programmers' reasoning, which is the focus of this study, must be based on empirical, explorative work.

In Section 2, we describe our long-range research goals and how they relate to this study. In Section 3, we explain the relevant concepts of concurrent programming and previous research on students' understanding of concurrent programming. In Section 4, we summarise the basic concepts of phenomenography and how it has been used in computer science education research.

In Section 5, we describe how we interviewed the students about a concurrent programming assignment and used phenomenography to distill the students' different understandings from the interview transcripts. In Section 6, we present these understandings and then, in Sections 7 and 8, we discuss how these types of understanding affect the students and what they mean for teaching concurrent programming and for developing ways to support students in developing correct concurrent programs.

## 2 Research questions

Our long-range goal is to help programmers create software that works correctly by aiding them in understanding, testing and debugging concurrent programs. We approach this by developing methods and tools to help programmers understand what a concurrent program does, especially when it is not working as expected. We intend to achieve this through visualisation of program execution and data.

Price et al. (1993) 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 note that when a 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.

Based on the above, our large-scale approach is to first identify the needs of the programmers and then design solutions to address them. The general questions we therefore seek answers to are:

- What defects do programmers introduce in concurrent programs, and why?
- Which of these defects are difficult to locate and why?
- What sort of visualisations (of a program execution or model checker counterexample) can assist a programmer in finding these most problematic defects, and how well do they work?

The intent of this study is to complement research into defects found in concurrent programs and visualisation of concurrent programs with information on different ways of thinking about concurrent programming. This will help when trying to reason about how a defect was introduced and when constructing visualisations to support programmers in understanding what their program does, especially when it does not behave as expected.

Naturally, this study can also be seen as part of the process of improving the teaching of concurrent programming by examining how students understand the concurrent programming concepts they have been taught.

## 2.1 Aims of this study

The purpose of the current work is to shed light on how programmers understand some core concepts, so that these insights can serve as a platform for exploring possible sources of errors, especially those that stem from insufficient or incorrect understandings. A better understanding of errors will help us make better debugging and development software.

We have chosen to focus on studying the understandings of students for three different reasons. Firstly, we assume that students, particularly master's and doctoral students, can be used as a source of information for designing tools and approaches that will also be useful for professional programmers. Secondly, we can collect and analyse large amounts of data from students, with less effort than from commercial software developers. Thirdly, an understanding of how students understand and approach concurrent programming is also important for improving how concurrent programming is taught. We briefly explain the relevant aspects of concurrent programming and previous research on how students understand it in Section 3.

In this study, we explore the different ways in which students in a concurrent programming course understand a concurrent data structure, the *tuple space*, as well as their various understandings of what developing and debugging a concurrent program means. We do this using an empirical, qualitative research approach called *phenomenography* (Marton & Booth 1997), which we will describe in Section 4.

## 3 Background

In this section we will briefly introduce concurrent programming as it pertains to this study and is perceived by experts (the textbook perspective, in Subsection 3.1) and students (the results of research into students' understanding of concurrent programming, in Subsection 3.2).

### 3.1 Concurrent programming

A *concurrent* program is a program that contains two or more *processes* that co-operate to achieve a goal, where each process is a set of sequentially executed instructions like a sequential program. These processes may actually be executing on separate hardware (multiple processors or execution cores in a single computers, or geographically separated computers). Alternatively, the concurrency may be simulated on a single processor by executing one process at a time for a brief period of time (Andrews 2000, Ben-Ari 2006).

Most workstation and server operating systems (e.g. Unix) provide support for dividing running processes between processors. Alternatively, a simulator can be used to simulate a multiprocessor machine for research or teaching purposes (Pears 1995).

Although a primer on concurrent programming is beyond the scope of this article, we will briefly describe the aspects of concurrent programming that are relevant to this study.

The most important distinction between sequential and concurrent programs is the inherently non-deterministic behaviour of concurrent execution; it is unknown how much of one process is executed during the time another one executes an instruction. The greatest challenge in writing concurrent programs is getting concurrent processes to reliably interact properly in the face of this nondeterminism (Andrews 2000, Ben-Ari 2006).

There are several approaches to ensuring correctness despite nondeterminism, including *deductive proofs* (usually manually constructed) and *model checking*, which can be based on a simplified computational model, such as in Spin (Holzmann 1997), or an actual programming language, e.g. Java PathFinder (Visser et al. 2003).

In order to co-operate, processes must be capable of communicating with each other. Many different mechanisms for interprocess communication (IPC) are available for this; only those relevant to this study will be described here.

One of the most common IPC mechanisms is *shared memory*: memory that can be written to by many different processes. This is typical of multi-threaded systems written in languages such as Java.

Other IPC mechanisms are typically provided to complement shared memory, some of which focus on preventing processes from proceeding with potentially harmful execution. The simplest is the *lock* or *mutex*, which allows the programmer to designate *critical sections* in a program, only one of which can run at a time. The *semaphore* can be considered an extension of the lock; it is essentially a shared non-negative integer value accessible through two operations: *V*, which increases the value of the semaphore by one, and *P*, which waits until the value of the semaphore is positive and then decreases it by one. A variety of *message-passing* mechanisms are often used in distributed systems; as the name implies, these involve processes sending messages to each other and receiving them either by waiting for a message or by buffering messages for later reading.

Gelernter (1985) describes, as a central part of the distributed programming language *Linda*, an interprocess communication mechanism called a *tuple space*. As its name implies, a *tuple space* consists of a space containing *tuples*, data records consisting of a *tag* (an identifier for the type of tuple) and (an ordered list of) zero or more data values. A tuple space can be accessed through three operations: *in()*, *out()* and *read()*. *out()* takes a tuple as an argument and inserts it in the tuple space. *in()* takes as its argument a pattern consisting of a tag and zero or more data values or formal parameters. As soon as a matching tuple is found, *in()* fills all the formal parameters with the corresponding values from the matching tuple, removes the tuple from the space and returns. *read()* behaves like *in()*, but does not remove the tuple from the space.

Tuple spaces can be used in many different ways. Therefore, they can be considered to be generalisations of several different IPC mechanisms. On the one hand, they are a form of shared memory; on the other, they can be seen as semaphores with associated data or a message-passing mechanism; all of these can easily be implemented using a tuple space (Gelernter 1985).

### 3.2 Students' understanding of concurrent programming

Ben-Ari & Ben-David Kolikant (1999) describe how high-school students' concurrent programming conceptions and working methods change during a course on the subject, focusing on difficulties faced by the students in dealing with concurrent programming. They found that students have problems with limiting themselves to operations permitted by the concurrency model, make assumptions based on informal concepts rather than use formal rules and avoid using concurrency. The students also applied development approaches that work well with sequential programming but not with concurrent programming, such as testing a program with a few representative inputs.

Ben-David Kolikant (2004) describes learning concurrent programming in terms of entering a community of computer science practitioners. The focus of her analysis is on the utterances of high school students while they are solving a concurrent programming assignment and the different perspectives on programming they represent. Specifically, she finds that the students, who have no programming experience but do have experience in using computer software, initially approach the concurrent programming assignment from a user's perspective, in which only the program behaviour seen through the user interface is taken into account. While one of the two students on which the analysis focused was able to switch to a programmer's perspective, allowing her to reason about synchronisation goals and possible interleavings and to systematically form a correct solution, the other continued to maintain a user perspective.

Hughes et al. (2005) state that even though many articles have been published on the subject of teaching concurrent programming, their review of two important forums for computer science education research uncovered no articles whatsoever on the subject of evaluating students' learning of concurrent programming. They argue that there is a need for quantitative empirical evidence.

## 4 Phenomenography

Marton (1981) notes that the world can be studied in terms of two different perspectives that he terms *first-order* and *second-order*. The first-order perspective involves examining and making statements about selected aspects of the world (*phenomena*), while the second-order perspective involves examining and making statements about how people experience these phenomena.

Marton argues that although educational research often focuses on the first-order perspective, the second-order perspective can also be fruitful in educational research. He notes further that second-order knowledge cannot normally be derived from first-order knowledge; we have no effective way to deduce how different people think about the world from what we know about it.

Marton also points out that people perceive concepts in many different ways. Booth (1992) describes, for example, the different ways in which students perceive recursion. Eckerdal & Thuné (2005) provide a recent example from the computer science domain: novice Java programmers perceive objects and classes in various ways.

Marton (1981) continues by describing the second-order research approach, which he terms *phenomenography*, as "research which aims at description, analysis, and understanding of experiences" and states that its focus is on understanding the variation in these experiences. The outcome of a phenomenographic re-

search project is thus a set of *categories of description*, where each category describes a qualitatively different way in which a phenomenon is understood in a cohort (Marton & Booth 1997).

Berglund (2006) describes the process of phenomenographic research in computer science education as consisting of a data collection phase and an analysis phase. In the data collection phase, the researcher interviews students about the phenomenon under investigation. The students are chosen with the intent of getting a diverse sample, in order to get a rich variation in their experiences of the phenomenon. Similarly, the interview must allow the student to express his understanding of the phenomenon of interest in many different ways. The interviews are then transcribed for analysis, during which the researcher looks for quotes that illuminate the students' various understandings and classifies quotes into categories of description. The analysis phase is typically iterative, with the tentative categories changing repeatedly as the researcher refines his analysis.

## 5 The study

In this section, we present the setting, how the data were collected and how the analysis was performed.

### 5.1 Setting

The students in this study participated in the Concurrent Programming course at Helsinki University of Technology during the autumn of 2006. All assignments were to be done in Java (version 1.4 or earlier). The assignments are described in more detail on the course home page<sup>1</sup>. Students could choose to do the assignments alone or in pairs; in either case, each group of one or two students submitted one solution that was graded the same way irrespective of the size of the group. As the students were required to submit their solutions through a WWW form that compiled their code, all submissions were valid Java programs.

The students were initially required to submit only their Java source code. In the event that their solution was rejected, they were required to submit corrected program code and reports explaining the reasoning behind the erroneous code and the steps they took to correct it.

In the first assignment, *Trains*, the students are given a simulated train track with two trains and two stations. Their task is to write code that drives these trains safely from one station to another using semaphores to co-ordinate the trains' movement.

The second assignment, *Reactor*, involves the Reactor pattern and its application to a simple multiplayer Hangman game. The students' task is to implement, using the synchronisation primitives built into Java, a dispatcher and demultiplexer for classic Java I/O, and to use this to implement a simple networked Hangman game that uses this Reactor pattern implementation.

The interviews focused on the third assignment, *Tuple space*, in which the students implement a tuple space using Java synchronisation primitives and use this to construct the message-passing section of a distributed chat server. The students' message-passing code communicates with the rest of the chat server system using method calls; a simple GUI front-end to the system is provided to the students for testing. The tuple space version in this assignment is a simplification of the original version: the `read()` operation has been removed, as it can be replaced

<sup>1</sup><http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml>

by an `in()` followed by an `out()` with the recently removed tuple without compromising the correctness of the operation. For consistency with Java naming conventions and clarity, `out()` and `in()` have been renamed `put()` and `get()` respectively.

## 5.2 Interviews

The first author conducted interviews with eight selected students regarding the Tuple space assignment of the Concurrent Programming course of Autumn 2006. The interviews were conducted between the announcement of initial submission results and the resubmission of failed assignments. The focus of the interviews was on the development process, especially the students' reasoning behind their design.

Twelve groups of students (nine students who did the assignment alone and three pairs who collaborated on the assignment) were selected for interview based on the assessments of their initial submissions for the third assignment. In order to maximise the variation of experiences based on the information available to us about the students, we chose groups with different types of problems with their code, as determined by the teaching assistant who graded the assignments. Ten out of 31 groups that failed the (initial submission of the) assignment and two out of 24 that passed the assignment (on their first try) were chosen and invited to an interview. Out of these groups, seven of the failing groups (six single students and one pair) agreed to participate.

The interviewees were allowed to choose the language of the interview: Finnish, English or Swedish. Five groups were interviewed in Finnish. Two East European students were interviewed in English, their primary language of instruction at our university. The two participating students who worked together on the assignment were interviewed together.

The interviews were semi-structured, i.e. they were in the form of a conversation using a set of prepared questions as conversation starters, and lasted from 30 minutes up to almost an hour. The questions were about tuple spaces, the design decisions made by the students in solving the assignment, their approach in determining whether their solution was satisfactory, and problems found by the students or the teaching assistant.

All of the interviews were recorded using a single table-top microphone and transcribed by the interviewer. The interviewer also wrote down the main points of the interview directly after the interview. These recordings and notes, as well as the code and documents submitted by the students and the teaching assistant's assessments of the students' submissions, form the source material.

## 5.3 Analysis

The analysis was done by the first author in discussion with the second author. Specifically, the authors first discussed the contents of two interview transcripts and then the iterative phase of the analysis was performed. In each iteration, the first author read through the transcripts looking for relevant quotes and formed categories based on these, building on the results of the previous iteration. The categories were grouped into outcome spaces by the issue they describe. The second author then examined these categories and made suggestions on how to improve them. The resulting categories from the last iteration are presented in the following section.

In the first iterations, the analysis focused on finding as many quotes as possible that illustrated ways in which the interviewees understood concurrent programming and approached the assignment. Initially,

quotes were grouped together if they essentially said the same thing (for example, tuples are, or are represented as, arrays). Then they were grouped together into tentative categories representing similar understandings of a phenomenon, which were grouped into tentative outcome spaces based on the phenomenon being discussed.

The categories changed in many ways during the analysis process. Starting from the third iteration, the emphasis of the analysis shifted to refining the preliminary categories. In some cases, only a few quotes regarding a phenomenon were found, in which case the data were deemed insufficient for further analysis. By the fourth iteration, only two outcome spaces remained in consideration as relevant for this paper; they are presented in the following section.

## 6 Results

In this section we present the outcome spaces of our phenomenographic analysis. In Subsection 6.1 we present the different *understandings of tuple spaces* that we found. In Subsection 6.2 we present the different *understandings of developing and debugging a concurrent program*.

Quotes are used to illustrate the categories. In these, the interviewer is denoted *Int* and the interviewees are assigned, to preserve their anonymity, the names *Evgeniy* and *Elena* (interviewed separately in English), *Filip*, *Fabian*, *Fritjof* and *Frans* (interviewed separately in Finnish) and *Freja* and *Fredrik* (interviewed together in Finnish). The quotes from the interviews in Finnish have been translated into English by the interviewer.

### 6.1 Tuple space

In this subsection, we present the different ways in which tuple spaces are described by the interviewees. This is summarised in Table 1.

#### 6.1.1 Specification

In this category, a tuple space is understood as a *specification*, i.e. as a *set of operations and how their inputs and outputs relate*.

An example can be found in the following extract from the interview with Evgeniy:

**Int:** Could you briefly explain what a tuple space does?

**Evgeniy:** There are two operations, to put a tuple in and... to... say, get a tuple in with a pattern... Uh, to get a matching tuple.

**Int:** If you have several matching tuples, which one do you get?

**Evgeniy:** Whichever, practically.

**Int:** And, if there's no match?

**Evgeniy:** Then, the execution suspends until there is one.

He explains what a tuple space is by referring to its definition. Fabian gives a similar view:

**Int:** So, what's the similarity [in the tuples] between different machines, given that you can't refer to the same variable?

**Fabian:** So, like, there are similar parts. There are like, the same, for example, they've been marked... The ones that, like, fit the pattern... It should match. And there are certain parts that match it and certain parts can be anything. Yeah, that's how

	Label	What is the tuple space described as?	What is in focus?	Framework
1	Specification	Operations on tuples	The properties of the operations	-
2	Implementation	Data structures and code	How a tuple space implementation works or could work	Part of a program
3	Usage	A tool to achieve a specific subgoal in a program	What a tuple space can be used for in a program	A program
4	Evaluation	A better way of co-ordinating distributed systems	The advantages of using the tuple space	Other communication and distributed data storage mechanisms

Table 1: Categories of tuple spaces

you get. Then you can directly mark what belongs directly, put some sort of identifier at the start or... that way, what you want to get from there. The message has the same identifier, then.

This statement is still based on the definition, but Fabian concentrates his explanation on the concept of a pattern.

We have in this category encountered an understanding that resembles that of a textbook definition, here exemplified by the following quote from Andrews (2000):

A process extracts a data tuple from TS by executing `IN("tag", field1, ..., fieldn)`; Each `fieldi` is either an expression or a formal parameter of the form `?var` where `var` is a variable in the executing process. The arguments to `IN` are called a template. The process executing `IN` delays until TS contains at least one tuple that matches the template, then removes one from TS. (ibid, p. 335)

Here, both Andrews and the students explain tuple space operations in terms of the operations on the tuple space and the inputs, outputs and delays of these operations.

Fritjof prefers to explain tuples in terms of programming language constructs rather than as abstract groups of values:

**Int:** Could you describe this tuple space? Like, what you put in it, what it does, what you get from it, sort of on that level?

**Fritjof:** Yeah. Uh... I don't have a fancy understanding of it, or one that is... necessarily entirely correct, but I'd say it's just like a set or a space containing unordered items. So, I don't know about these tuples, but I'd imagine, or I like to think of them as sort of arrays of some sort of elements, so, for example, a tuple is some *n*-element table in there.

Here, Fritjof uses programming language constructs such as arrays, but is still describing the tuple space in terms of its interface (in this case, the data format used to communicate with it).

Fredrik explains pattern matching in the tuple space in similar terms:

**Int:** Yeah, how do you choose what to get, for example?

**Fredrik:** It's quite...

**Freja:** Isn't it kind of like getting with parameters that are 'identifiers' for these tuples? So they sort of have an identifier.

**Fredrik:** Right, right, right, and, the amount of fields or attributes and... also null values.

The statements of Fritjof and Fredrik are inspired by the requirements of the assignment, which provided a very specific definition of tuple space operations and Java-specific definitions of how tuples and patterns are represented:

```
public interface TupleSpace
```

- `public String[] get(String[] pattern)`

Remove and return a tuple (an array of entries) matching `pattern` (which may not be `null`) from tuple space. Block until one is available. A tuple matches a pattern if both have the same amount of entries and every entry matches. A `null` entry in the pattern matches any object in that entry in the tuple. Any other object `p` in the pattern matches any object `t` in the corresponding entry in the tuple for which `p.equals(t)` (i.e. contains the same character string). If several matching tuples are found in the tuple space, any one of them may be returned.

The returned tuple must have exactly the same textual contents as the tuple that was put (contain the same amount of `String` objects as the original and each `String` equals the `String` in the same position in the original), but may be a different array object and may contain different `String` objects).

- `public void put(String[] tuple)`

Insert `tuple` in tuple space. `tuple` is an array of any length greater than zero and is not `null`. `tuple` may not contain `null` values. Tuples stored in the tuple space must remain unchanged as long as they are in the space.

Thus, the two interviewees discuss what a tuple or a pattern is, based on the definition given in the assignment<sup>2</sup>.

In this category, the tuples per se are in focus. Different aspects of them can be highlighted, such as operations on them (Evgeniy) or the structure of the tuple (Fritjof) or both (Fabian). The specification might have its roots in the theoretical specification (Fabian) or from the assignment the students were to solve (Fritjof and Fredrik).

According to phenomenographic theory, when someone experiences something, some aspects of the experienced phenomenon stand out in the fore, while other aspects of it or other contextually related phenomena reside in the background (Marton & Booth 1997). However, this category shows a slightly different structure: as the tuples are seen in isolation,

<sup>2</sup><http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml>

in a decontextualized manner, they are experienced not against any background, but as atomic objects in their own right.

### 6.1.2 Implementation

In this category, a tuple space is understood *as it is implemented*.

Let us listen to Fritjof, for example:

**Int:** ... and the get operation does what?

**Fritjof:** So, if you call `get()` with a certain pattern, something, you want from there a specific tuple; it looks through the tuple space for such a set or item. If it finds it, it returns it directly, immediately. If it doesn't find, then it actually waits for someone to put an item there with the `put()` operation.

This way of understanding tuple spaces also allows the large-scale structure of the implementation to be described, as in Frans's statement:

**Int:** OK, right, and how does this transfer affect the tuples, then?

**Frans:** Those tuples are somewhere in a central place, so that means that if you get something from there then it isn't there anymore. It should, uh, take into account that it... If there's some tuple there and somebody gets it from there, then nobody else can get it from there before that somebody has returned it there.

His statement illustrates a problem with this way of experiencing tuple spaces: the implementation need not be the same everywhere.

Fredrik starts his description of a tuple space in the following way:

**Int:** So how does it [the tuple space] work, then? How is it used?

**Fredrik:** It's a data container in which it has been ensured that you don't read and write to it at the same time through synchronisation.

Fredrik's statements, that seemingly express a high-level understanding, actually reflect his own implementation of tuples in the assignment. There, he relies on a single lock (implemented using the Java `synchronized` keyword) that ensures that only one operation at a time is performed on his tuple space. This, in turn, ensures that the tuple space operations behave atomically, as specified.

This category extends the previous, as the implementation is written to match a specification, or at least is written to achieve a goal. In any case, the externally visible behaviour (which the specification describes) can be deduced from the implementation.

### 6.1.3 Usage

This category describes a tuple space as a *data structure or a module that can be used as a part of a program in order to achieve a specific subgoal in a program*.

This is illustrated by Frans's answer, when asked to explain how using a tuple space affects the program:

**Int:** OK, so they're [the tuple spaces] intended for distributed systems that may have a common space for many machines

or processes, despite not having variables in common. How does this affect them?

**Frans:** Um, what are you getting at? Apparently, with the help of the tuple space you can implement some sort of monitor or something, with which you can...

He states that synchronisation is the purpose (or at least one purpose) of using tuple spaces. Clearly, this statement refers to usage of the concept within a program.

Fritjof explains how his chat system uses a tuple space to get unique message numbers over several distributed processes:

**Int:** So, how does it [your implementation] work when it's on several machines?

[...]

**Fritjof:** I put these specific tuples there, that always stay the same, so, like, even though the message counter, that is... that is, that way you keep track of those messages that are put in the tuple space, so we have this single message counter tuple there, and with the aid of that, uh, the remote machines sort of synchronise their functioning, so the counter is fetched from there. One machine gets the counter, then another machine, even though it's trying to get the message counter tuple at the same time, it doesn't find it there. And... And when the first machine has processed the tuple, got the value from there, it puts the tuple back in there and the other machine can then get it from there.

In this case, he is describing the *implementation* of a chat system that *uses* a tuple space for communication. The characteristic of this category is how a tuple space is used, or its purpose in a program. This implies a broader perspective than that of the previous category, since the tuple space must be seen in the context of a program for its usage within a program to be seen.

### 6.1.4 Evaluation

Here, the tuple space is seen in terms of the *advantages of using it in contrast to other data structures or message passing mechanisms*.

When asked how using a tuple space affects the programs using it, Elena answered:

**Int:** OK, so if you have a distributed environment here, where you have one pro... two different programs that might be in completely different machines with a tuple space, and, uh...?

**Elena:** It makes, uh, the communication between them; it makes it very much easier, so, um, between different... ah, implementations, they can communicate with each other by way of this pool.

Filip compares the tuple space with a semaphore:

**Int:** Could you explain what a tuple space is; how it behaves in general...?

[...]

**Filip:** It's kind of like an improved version of a semaphore, so, in a semaphore, like, you've got to know in advance what the semaphore is connected to, but the tuple, you can attach information to that. But it... It's like sort of... A semaphore is a special case, you can also use a tuple in such a way that it either has a flag set or not.

Here, he is contrasting the tuple space with a semaphore, and noting that a semaphore can only indicate that something has happened (typically, that a resource is available), while a tuple can contain additional information (such as the contents of a message).

The purpose of using tuples as building blocks of a program, which was in the fore of the previous category, is taken for granted here. Instead, the benefits of using tuple spaces are seen in relation to other ways of communicating.

## 6.2 What does it mean to write and debug a concurrent program?

In this subsection, we present the different ways in which the interviewees understand the process of developing and debugging their program. This is summarised in Table 2.

### 6.2.1 Implementation

In the first category, writing and debugging a concurrent program means *making it run*; the *coding* itself is the focus.

The programming of a complex sequence of events can be experienced in this way. Fritjof explains his message-writing implementation:

**Int:** So, how did you fix it [a race condition in using the message counter]?

[...]

**Fritjof:** Anyway, the idea is that, like, in that `writeMessage()`, uh, I right at the start call, uh, that it fetches the message counter tuple... and its... method that it's fetched with, it only removes the tuple from there... the counter tuple, so, it, like, fetches it for itself... So, this time, it doesn't, like, do those `put()/get()` operations in the same method, so it just takes them from there, and then after that... After that it, uh, writes the, uh, client's message, or makes a tuple that is the client's message, and uses the counter that it fetched. And, uh... The message is always put in the tuple space. After that, uh, the message ID is incremented, which, of course, is for every, uh... chat channel separate and not until that is done, finally, the tuple is put back in the tuple space.

Java constructs (e.g. methods, calls, incrementation) and the roles in which they are used (e.g. tuples, counters) are taken for granted and constitute the building blocks from which his explanations are built.

While Fritjof discusses programming, we can hear a similar discussion from Evgeniy when he discusses debugging:

**Int:** In what did the first attempt...?

**Evgeniy:** The first attempt uh... had just the backlog of messages in the tuple space... Uh, where, with a, I th... I don't remember was it counters or something where channel listeners would pick a tuple from the space and return it until the counter is zero and then discard it instead, but, uh, for some reason; for some obscure reason I don't understand, still, it didn't work, uh, when there were more than one channels, uh, more than one listeners or with, with load it started to go all bad and the... tuple started to disappear.

Evgeniy explains how an early version of his program misbehaved in low-level terms. Although he does not use language containing Java constructs, his description is worded, as Fritjof's above was in part, in terms that correspond to specific Java constructs. For example, Evgeniy's counters are integers stored as fields in tuples of a specific form.

Elena also takes her point of departure in the program itself, when asked to explain her chat system design:

**Int:** Was this design the first one that occurred to you or did you consider some other way of doing it?

[...]

**Elena:** The basic idea, however, I can, so when, ah, when, uh, when, uh, the method `writeMessage()` of the server is called, what it does is that it gets the... the number, because there's a tuple which keeps the maximum number of servers.

[...]

**Elena:** New, newcoming servers see the... uh, get the tuple and update it with an increased number...

In this quote, Elena specifically takes her point of departure in the program execution at the level of the programming language when she describes how data must flow between different servers.

The three students whose quotes have served to illustrate this category have all explained writing and debugging of programs in terms of the implementation of the programs. The core of this category is the execution of the program, and descriptions at the level of Java code, regardless of whether the students worded their explanations in Java terminology. This is seen against a background delimited by the language features that can be used in a certain situation. In other words, the reasoning does not extend the program and its execution in any way, other than assigning roles and purposes to the Java language constructs.

### 6.2.2 Solving technical problems

In this category, writing a concurrent program is experienced as *solving technical problems*.

Let us listen to Filip, for example:

**Int:** So, how is this information transferred [between machines], then, roughly?

**Filip:** Well, it remained kind of unclear to me how it would be done in a practical application, because, because, uh, one should, like, be careful that the tuples are always the same on all machines and then, if somewhere an acquire is done, then the information is transferred to them all before they can do anything at all [...] to the corresponding tuple.

[...]

**Filip:** Like, that all the, sort of, servers have to know the same, like, tuple space. They've got to have all tuples known to everyone.

He raises the issue that the tuple space is a data structure intended for distributed computing and that the different computers involved must co-operate to ensure that operations performed on the space on one computer are visible on all other computers. He discusses this in terms of a technical system, that is, he describes 'what happens' and 'how it functions' in a

	Label	What is developing and debugging described as?	What is in focus?	Framework
1	Implementation	Writing and debugging code	The code and its execution	Relevant programming language constructs
2	Solving technical problems	Finding solutions to a series of technical problems	Central ideas of concurrent programming	The program, seen as a technical entity
3	Producing an application	Finding solutions to real-life problems	What users need from the program	Context in which program is used

Table 2: Categories of developing and debugging

different way from that expressed in the previous category, where the focus is on language constructs as primitives.

Similarly, when asked to elaborate on his reasons for choosing a particular chat system design, Fritjof notes:

**Int:** Why'd you choose this particular solution?

**Fritjof:** ... a 'send copies to everyone'-style solution first came to mind, but, uh, then it... Then it came to me that it isn't really clever to do it this way, so... So, one, only one copy should be stored at a time.

Frans takes his point of departure in similar concepts:

**Int:** Do you have any idea what messages it can trans...leave undeleted?

**Frans:** When each active listener is sent the messages, could it be that someone... some listener, some active listener leaves before it's read all the messages sent to it?

When asked why his system may incorrectly leave some messages indefinitely in the tuple space (a form of memory leak), Frans answers in terms of messages being sent to listeners, the large-scale behaviour of the system, rather than the underlying tuple space operations.

Here, we have seen a way of experiencing writing and debugging programs in which the task is seen as solving a series of technical problems. This is viewed and expressed in terms of general and central ideas of concurrent programming and the system that is constructed. This can be contrasted with the previous category, in which the basic entities are the programming language and the constructs expressed in it. Thus the current category has a broader perspective.

### 6.2.3 Producing an application

In this final category, the programming task is understood as *solving problems relevant for a usage context*.

Filip demonstrates this viewpoint when responding to the teaching assistant's complaint that his solution does not allow messages to be repeated or be empty:

**Int:** So you mean you've planned your solution to sometimes duplicate messages and then compensated by deleting them later?

**Filip:** Yeah, I figured the duplicate removal didn't hurt, especially since it doesn't happen randomly, but when a new user joins

the channel. Then, uh, the empty messages, that's just that I've assumed that you don't want to put empty messages. It's, both in the transmission and reception, been tested whether an empty message has come, but this was also an error that led to failing the assignment.

As opposed to the previous categories, the technical concepts here become tools for solving a real-life problem. The student is no longer working to implement a specification; he is trying to meet the needs of the users. The background is therefore no longer a purely technical context (the program); it is the real-world situation for which the program is intended. In addition to the system-level technical concepts of the previous category, the basic entities now include user requirements and desires.

## 7 Discussion

In this section we will examine the categories presented in the previous section from different perspectives. First we will discuss the meanings of the different categories from an educational point of view. We will then discuss what the categories mean for our long-term research.

### 7.1 Tuple spaces

Bloom's taxonomy of educational objectives (Bloom 1956) is widely used in higher-education course design to ensure a proper balance between rote learning and high-level skills such as synthesis and evaluation. However, its applicability to computer science teaching is debatable, as the goal of computer science is often perceived by teachers to be application. (Johnson & Fuller 2006)

Johnson & Fuller (2006) therefore propose a revised Bloom taxonomy which retains *knowledge*, *comprehension* and *application* in their original order, places *analysis*, *synthesis* and *evaluation* as equals above the lower three and adds *higher application*, application informed by analysis, synthesis and evaluation, at the top. The revised taxonomy is particularly relevant when computer science is approached from an engineering perspective where application is clearly the goal. This motivates a comparison with our tuple space categories, in which students understand a concurrent programming concept in terms related to different skills and tasks.

The specification category is essentially knowledge; the students are explaining the tuple space description given to them in the textbook, lectures or assignment specification.

In the implementation category, the students have constructed their own implementation, which at least

requires application of the specification and concurrent programming in Java.

The usage category is another form of application, except here the tuple space knowledge is being applied to achieving a program's goals using a tuple space. Finally, the evaluation category clearly corresponds to evaluation.

The tuple space categories therefore span a large portion of the revised Bloom taxonomy, although analysis and synthesis do not appear. Although lack of evidence is not evidence of lack (especially since only students who failed the assignment were interviewed), this does raise the question of whether the teaching of concurrent programming can be improved by encouraging students to analyse and synthesise.

## 7.2 Development

Software engineering emphasises ways of managing complexity and quality that rely on different perspectives on the software that is being developed. The categories of developing that we found are similar to several of the different views needed in many common software development processes.

The implementation categories of both developing and tuple spaces are obviously necessary for practical software development, in which formulating an implementation in a programming language is essential. The solving technical problems category corresponds to design. In this assignment, what the students are performing is essentially module design, as the specification the students are provided with is more or less a finished architecture design. This specification is also seen in the specification category of tuple spaces. The design of the chat application also involves usage of tuple spaces; evaluation is not necessary in the assignment, as the use of tuple spaces was mandated.

These first two categories of developing can also be considered facets of what Ben-David Kolikant (2004) calls the programmer's perspective, which includes reasoning in terms of both the concurrency model and the implementation.

As the assignment is to perform the low-level design and implementation of parts of a system with clearly specified requirements, we did not expect any consideration of the user of the system. The application production category, which is suited for requirements analysis, is therefore unexpected in this study. As seen in Subsubsection 6.2.3, it can become a hindrance in programming assignments such as the one in this study. Adherence to the specification is considered by the teaching staff to be the goal of the assignment. This leads to problems when the student moves beyond the specification to address perceived user requirements instead. This also demonstrates that having a skill or understanding is not enough; the student must also learn to use it in the appropriate context.

Each one of these categories is suitable for some task in software engineering. Taken as a whole, these three categories more or less span the perspectives necessary for a full software development process. This suggests the idea of teaching students requirements analysis for concurrent programs instead of keeping the teaching and assignments on the design and implementation levels. This could also help students understand the appropriate situations in which to use each skill.

## 8 Conclusions

In this paper we present different categories of understanding tuple spaces and developing a concurrent program. We describe the relationship between

these categories and the skills we want students to develop. We then discuss how this information can be used in developing concurrent programming education, by exposing students to different views of the same phenomena. We also discuss the use of this study in our long-range research in supporting concurrent programming, first by helping develop a model of how bugs are introduced in concurrent programs, and second by developing software to display information about these programs in a fashion consistent with the programmer's understanding.

### 8.1 Long-term research impact

In the previous section we concentrated on an educational perspective on the results of this study. We will now consider our results in terms of our long-range research questions, as described in Section 2.

Different errors can be the result of completely different ways of thinking. In some cases, such as the application production category described in the previous section, approaching a problem from the wrong perspective may lead to erroneous conclusions. In other cases, the nature of the errors depends on the perspective or task at hand. Thus understanding how the programmer is thinking is important in finding ways to prevent errors from being made as well as determining the errors to look for in verification. For example, if a programmer misunderstands the requirements or specification of a system or module, he will be also be testing according to his erroneous understanding of the requirements. One way to address this is to include test cases in the specification, providing both clarifications to the programmer and test cases that are not dependent on his understanding.

This is one way in which this study supports our long-range goals: as software defects are the results of programmer error, i.e. incorrect thinking, it is necessary to understand how programmers think in order to construct a model of errors. For example, goal-plan analysis (Spohrer et al. 1985) provides a way to identify and categorise bugs based on a model of the problem-solving process. For each goal, there are one or more plans for achieving it, each with several sub-goals; this is expressed as a goal and plan (GAP) tree. Spohrer et al. (1985) infer the GAP tree from the programs. Knowledge of how programmers understand their development process is both a source of possible plans and empirical support for the plans identified from the programs.

Another way in which the results of this study can be used in our research is in designing useful visualisations of programs and their execution. Debuggers traditionally focus on code, as in the implementation category. However, the solving technical problems category suggests an alternative perspective on debugging: that it would be useful to provide supporting tools, such as execution visualisations that show program behaviour in ways that support the user's understanding. This could be done by allowing the user to group together parts of the code or execution to correspond to his understanding, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. (1993). The tool would then visualise the behaviour of the program in a fashion closer to the programmer's view. For example, if the programmer sees his program as a set of communicating entities, the tool should be able to show him 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.

## 8.2 Future analysis of the data

The semi-structured format of the interviews allowed the interviewees to express themselves on a wide range of subjects related to concurrent programming and the associated teaching and assignments. One possible topic for future analysis of these interviews is the types of errors students make and the factors that contribute to them. Another is how students approach and understand the learning of concurrent programming and the testing and debugging of concurrent programs. It is also possible to examine the categories presented in this article in greater detail, examining different conceptions and misconceptions within each category.

## References

- Andrews, G. R. (2000), *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley.
- Ben-Ari, M. (2006), *Principles of Concurrent and Distributed Programming*, second edn, Pearson Education.
- Ben-Ari, M. & Ben-David Kolikant, Y. (1999), Thinking parallel: The process of learning concurrency, in 'Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education', Cracow, Poland, pp. 13–16.
- Ben-David Kolikant, Y. (2004), '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.
- Berglund, A. (2006), 'Phenomenography as a way to research learning in computing', *Bulletin of Applied Computing and Information Technology* **4**(1).
- Bloom, B. S. (1956), *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*, Addison-Wesley.
- Booth, S. (1992), Learning to program: A phenomenographic perspective, Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden.
- Eckerdal, A. & Thuné, M. (2005), 'Novice Java programmers' conceptions of "object" and "class", and variation theory', *SIGCSE Bulletin* **37**(3), 89–93.
- Gelernter, D. (1985), 'Generative communication in Linda', *ACM Transactions on Programming Languages and Systems* **7**(1), 80–112.
- Holzmann, G. (1997), 'The model checker Spin', *IEEE Trans. on Software Engineering* **23**(5), 279–295.
- Hughes, C., Buckley, J., Exton, C. & O'Carroll, D. (2005), 'Towards a framework for characterising concurrent comprehension', *Computer Science Education* **15**(1), 7–24.
- Johnson, C. G. & Fuller, U. (2006), Is Bloom's taxonomy appropriate for computer science?, in A. Berglund & M. Wiggberg, eds, 'Proceedings of 6th Baltic Sea Conference on Computing Education Research, Koli Calling', Uppsala University, pp. 120–123.
- Marton, F. (1981), 'Phenomenography — describing conceptions of the world around us', *Instructional science* **10**, 177–200.
- Marton, F. & Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associates.
- Pears, A. N. (1995), Using the DiST simulator to teach parallel computing concepts, in 'International Forum on Parallel Computing Curricula', Wellesley, Massachusetts.
- Price, B. A., Baecker, R. M. & Small, I. S. (1993), 'A principled taxonomy of software visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.
- Spohrer, J. C., Soloway, E. & Pope, E. (1985), 'A goal/plan analysis of buggy Pascal programs', *Human-Computer Interaction* **1**, 163–207.
- Visser, W., Havelund, K., Brat, G., Park, S. & Lerda, F. (2003), 'Model checking programs', *Automated Software Engineering Journal* **10**(2), 203–232.

# Applying Creativity in CS High School Education – Criteria, Teaching Example and Evaluation

**Ralf Romeike**

University of Potsdam  
Department of Computer Science  
A.-Bebel-Str. 89  
14482 Potsdam, Germany  
romeike@cs.uni-potsdam.de

## Abstract

This paper describes an innovative method for teaching computer science in general high school education, illustrated with the example of introductory programming. Analyzing the literature in CS education research we found that creativity is rarely regarded, especially in high school education; although a few authors describe promising results from applying creativity. We designed and applied a framework for designing creative CS lessons based on a set of creativity criteria. The conducted teaching unit on introductory programming fulfilled the expectations: the students learned with high motivation and interest, the learning objectives were met and the students' picture of CS improved

**Keywords:** Programming, creativity, learning, teaching, motivation, high school computer science

## 1 Introduction

Computer science nowadays has taken an important position in German high school education and is represented as a mandatory or elective subject in almost all secondary schools<sup>1</sup>. The role of the subject is not to educate young computer scientists or programmers, but to provide the students with a positive attitude towards IT systems and a confident, responsible use of IT in the information society, and to allow the students an insight into the science itself. Even though the students arrive being more and more familiar with computers and with a general positive attitude towards them, CS has to deal with problems: low motivation, decreasing interest in the 'core' fields of CS such as programming, low grades, low participation of female students and the transfer of a wrong image of CS in schools. This in continuation has an additional impact on CS studies at university: students

often enter with a wrong perception of CS and drop out early (Romeike & Schwill 2006). These problems stand in sharp contrast to some out-of-classroom observations where students and professionals spend a large amount of their free time dealing with programming or other aspects of CS. A key factor for engaging with programming seems to be creativity. In a study about the motivation of open source programmers, creativity-related factors were found to be the most pervasive drivers (Lakhani & Wolf 2005). In an interview with an outstanding motivated student of CS, creativity was also named as the most important factor for engaging in programming (Romeike 2006). The school subject of CS, as we see it, is strongly connected with creativity and can make use of it in manifold ways. Being creative fosters motivation and interest in the field, the subject of CS offers a fertile ground for creativity as the concepts and tools are well understandable and structured, and the omnipresent IT is beneficial for creativity (Romeike 2007c, Shneiderman 2000, Clements 1995, Thomas et al. 2002). One possible way that creativity can be applied in the classroom is described in this paper. After defining creativity and the consideration of it in previous research in CS education, we describe criteria for designing creative CS lessons. Based on these criteria a creativity framework is presented and applied in a lesson example for introductory programming, which was performed and evaluated in a German high school. The evaluation and the results are discussed.

## 2 Creativity

The term creativity is used with different meanings and is discussed controversially in psychology. Common speech usually defines something as creative when it comes from the arts or is something extraordinary. But not only artists can be creative. Everyday life requires creativity – and so does CS. There is agreement in psychology that something is creative if it is new, original and useful. How can an educator expect new and original achievements from his students? Boden (1990) describes two aspects of creative achievements. Historical creativity (h-creativity) describes ideas that are novel and original in the sense that nobody has had them before. Something that is fundamentally novel to the individual Boden describes as psychologically creative (p-creativity). In an educational context the latter is more interesting and can be aimed for in the classroom. Thus the difference between an exceptionally creative person and a less creative person is not a special ability. It is

---

<sup>1</sup> In the German education system secondary education starts – depending on the federal state – with the 5<sup>th</sup> class (age 10) or 7<sup>th</sup> class (age 12). Students aiming for the Abitur, which is a prerequisite for higher education, attend the secondary II finishing after 12<sup>th</sup> or 13<sup>th</sup> class. CS lessons are offered to students starting from 7<sup>th</sup>, 9<sup>th</sup> or 11<sup>th</sup> class.

based on a larger knowledge in a practical and applied form as well as on the will to acquire and use that knowledge. With that in mind, in this paper we call something creative if it leads to personal new, unique and useful ideas, solutions or insights (cp. Runco & Chand 1995, Kaufman & Sternberg 2007). As summarized by Fasko (2000), in the classroom creativity can enhance learning through improved motivation, alertness, curiosity, concentration and achievement.

### 3 Creativity in CS Education

Computer science, as computer scientists see it, is a creative field to work in (e.g. Leach 2005, Glass 2006). Hence it is astounding that creativity is rarely reflected in CS education research. Even today, a search of the keyword ‘creativity’ in the ACM Digital Library returns only a few papers related to education. These papers can generally be assigned to a few groups in the contexts of problem solving, problem finding, motivation, and improving lessons and ICT to support creative practice.

Scragg et al. (1994) argue that CS is a fundamentally creative endeavor. Students need to be encouraged to discover insights in the creative process of problem solving. Hill (1998) describes open-ended problem solving and design processes in technology education as creative processes that engage exploration. She suggests moving away from making models to making prototypes for real-life contexts. In contrast, popular concepts for the school subject of CS are focusing especially on making models and yet leave out their implementation (Hubwieser 2000).

Some authors call attention to the field of problem finding/posing/identifying, which involves creativity and is important in the field of computer science. In the lesson context it is not limited to finding completely new problems, but includes also reformulating given or existing ones (Lewis et al. 1998, Kaasbøll 1998). Sutinen and Tarhio (2001) suggest it is better to speak about problem management than problem solving, as computer experts need skills that include problem recognizing and formulating.

A case study on the use of game programming in CS education was performed by Long (2007). She found that “being able to solve problems on my own” and “to be able to be creative” were the most important factors influencing intrinsic motivation.

Gu and Tong (2004) found, in an empirical study, that in software development courses the students perceived architecture design and programming as creative and that these phases were preferred. For similar reasons some authors employ creativity as a factor for raising motivation and interest in CS lessons. This was done by, amongst others

- changes in the environment and encouraging creative, hands-on learning and exploration into the projects in a data structures and algorithms course (Lewandowski et al. 2005)
- letting students choose and process their own problems (Meisalo et al. 1997)

- allowing programming as personal creative expression (Peppler & Kafai 2005, Resnick 2002)
- presenting programming in an entertaining discovering way (Wilson 2004)

Resnick (2007) sees creative thinking skills as critical for success and satisfaction in today’s society. He reports that in computer clubhouses a creative use of the computer and programming is learned by promoting to students a spiral cycle of imagining, creating, playing, sharing, reflecting, and back to imagining. This he describes to be ideally suited to the needs of the 21st century.

Some researchers report achieving a positive effect on students’ performance by applying creativity techniques in CS courses (Epstein 2006) or using creative methods for teaching programming (Chaytor and Leung 2003).

Several authors in CS education call for creativity, because

- Graduates in CS are missing creativity and problem-solving skills (Mittermeir 2000)
- Creativity is underrepresented in the curriculum (Sweeney 2003)
- Women drop out because there is no room for individual creativity in CS courses (Guzdial & Soloway 2002)
- Creative abilities are seen as the highest form of literacy, including computer literacy (Van Dyke 1987)

Computers have been found to be a fertile tool for supporting creativity. Many articles address IT support for creative practice, however there are just a few related to computer science education in schools (e.g. Clements 1995).

In summary these works show a broad spectrum of examples where creativity was identified to be beneficial and where creativity was successfully applied for enhancing learning. It is therefore quite surprising that the opportunities offered by creativity are not more frequently applied in general computer science education. It seems promising to us to investigate what the application of creativity can do for high school CS education.

In an analysis of the relevant literature we investigated the application of and the possibility of creativity in published computer science lesson examples (Romeike 2007a). We found that creativity was rarely employed. However, the lessons analyzed offered chances to do so and could be extended to creative lessons when considering creativity factors. Apparently teachers even partly prefer non-creative students, as they are easier to handle in the classroom<sup>2</sup>. Such a teacher attitude encourages students to prefer familiar ways that seem

---

<sup>2</sup> “Anyhow I am afraid that students let their creativity play too much so that the results of this project would be of limited usefulness.” (Janneck 2006).

safe and risk free but do not leave much space for creativity<sup>3</sup>.

We consider the role of creativity in CS from two perspectives. First, we believe that creativity is essential to CS. Second, CS makes it easy to be creative. Keeping in mind the relevance of creativity for CS and its value for CS education may help educators to overcome some common problems they experience in the classroom.

Formulated creativity criteria will help teachers in planning lessons and regarding the creativity potential that CS offers.

## 4 Criteria for Creative CS Lessons<sup>4</sup>

To obtain a foundation for creative CS lessons, we set up a catalogue of criteria based on findings in the literature of psychology and education. These criteria can be used for designing and evaluating computer science lessons. They reflect and combine general pedagogical principles that are essential and beneficial for creative practices in CS education. In addition they consider typical tasks and principles that are common in CS.

### 4.1 Requirements for the Subject

**Relevance.** We define the subject of a lesson as the topic that is used for illustrating the teaching matter. As creativity requires personal involvement it needs to be appealing and thus relevant to the students, or needs to be presented that way.

**Problem management or creation of a product.** Gardner (1993) classified five types of creative activities. Two of those are typical for CS and should be aimed for in creative lesson phases: problem solving and the creation of a product. This includes the implementation of a model, not just the finding of a theoretical solution.

### 4.2 Requirements for Tasks

**Subjective novelty.** This important criterion for creativity is often overlooked by teachers who use tasks very similar to those that have been discussed in detail in the lesson. Even if it is unlikely that a student will come up with a general new solution or product, subjectively new (p-creative) ones should be aimed for.

**Openness in possible results, approaches and solution methods.** Creative processes are characterized by aspects of problem finding and creative problem solving, exploring and discovering. This is possible only if tasks allow several approaches to the problem, diverse

solutions, and solutions that can differ in the degree of elaboration.

**Application of concept knowledge.** A solid foundation of knowledge is essential to creative practice. In a creative lesson phase, concept knowledge needs to be emphasized in contrast to product knowledge or factual knowledge.

**Inspiration.** A creative achievement is always preceded by a stimulus. Type, content, formulation or circumstances of a task and learning situation can provide such an initiation. For CS lessons this includes revealing to the students, for example, what a piece of software will be used for and which 'broader' problems it is supposed to solve.

### 4.3 Student-oriented Requirements

**Identification.** Creative practice may get a person enthused, getting him or her deeply involved with a task, and may trigger a flow-condition. Fundamental for this is that the person can identify himself with the task. For CS lessons this implies that the content needs to be (or can become) meaningful to the student, e.g. by taking over responsibility and/or later presentation.

**Originality.** Every student is a unique individual with his or her own ideas, visions and preferences. Obeying this criterion means allowing space for a student's originality demands, i.e. letting the student bring in a personal touch.

### 4.4 Requirements for the Teaching Environment

**Experimenting.** Being creative means to experiment with ideas, to explore the space of possibilities and to test solution possibilities. A tool used should provide meaningful feedback; for example, the compiler of a programming environment supports experimenting in CS lessons as it gives detailed feedback to the learner.

**Freedom in time.** Creativity is hard to realize under time pressure, as time is needed to gather, evaluate and realize ideas. Projects in CS lessons support this criterion.

**Climate of diversity.** Group pressure, early evaluation and expected perfection are known to oppress creativity. Instead the lesson should allow encouragement and inspiration among students. New ideas should be welcome and diverse solutions supported and presented.

**Teacher as a coach.** The teacher needs to diminish the leading role of transferring knowledge, correcting and assessing. Instead the teacher assists only where a problem cannot be solved by a student himself. He motivates and encourages the students.

## 5 Introduction to Programming by Applying Creativity Criteria

The question of how programming should be introduced is a central issue of computer science classes in schools over and over again. But universities as well as schools struggle to provide students with a smooth transition into the field of computer science. Often these introductory

<sup>3</sup> Taking risks is difficult for creative students because creativity is not always rewarded with good grades (Sternberg & Lubart 1991). Perhaps this is due to the negative attitudes teachers hold towards creative students, as evidenced by the findings of Westby and Dawson (1995).

<sup>4</sup> A detailed derivation and explanation of the criteria were performed in (Romeike 2007a).

courses and topics are found to be the cause for computer science being seen as hard, mechanistic or even uninteresting or discouraging (Bergin 2005, Curzon 1998, Mamone 1992, Rich et al. 2004, Tharp 1981, Feldgen 2003). We believe it does not need to be this way. The chance to develop software using a programming language can nicely demonstrate that computer systems can be shaped by the student in a motivating way.

For the realization of a lesson example which is motivating and encouraging for the students, and at the same time allows for learning about computer science close to the subject, the criteria for creative computer science lessons were regarded and applied. The lesson example was designed for introducing an 11<sup>th</sup> class of computer science in a German high school to programming. As a programming language and creativity supporting tool the visual programming language Scratch (Maloney 2004) was used. The application of the creativity criteria results in a creativity framework that was followed in the teaching unit and ensured that all of the criteria could be given due regard. The framework is described as follows, and illustrated by details of the lessons. The teaching unit in detail can be found at (Romeike 2007b). The educational objectives of the teaching unit are summarized in Figure 1.

## 5.1 The Creativity Framework

### 5.1.1 Motivation for New Concepts of Programming

Motivation is an essential part of teaching. Receiving students' attention and fostering motivation was supported by showing the use and relevance of the contents to the students and by choosing topics that are meaningful to them, e.g. animating their name or a story of their everyday life or imagination, and the development of games that can be played by them. Often new concepts were brought up by the students themselves after discovering and applying them in their projects before they were formally introduced.

### 5.1.2 Laying out the Fundamentals

The introduction of new content was done by applying a building block metaphor. Attributes and uses of the programming concepts in the Scratch programming language were discovered or explained. Beneficial for this view is the visual representation of CS concepts in Scratch as blocks that can be snapped together. In this way students learn an appropriate visual representation of the concepts and do not have to deal with syntax errors, as they are not possible. As a teaching method, the concepts were introduced either by the teacher, by work sheets or by student presentations.

### 5.1.3 Inspiration

It is essential for a creative lesson to provide an inspiration to the students, generally by showing an example program or brainstorming about possibilities. This allows the students to spark their creativity, to balance what they may want to achieve and what they

can achieve with the concepts learned so far and what the programming language is capable of.

### 5.1.4 Challenging the Students

Challenging the students was done with open-ended tasks with variable solution complexity and independent working time for the students. The tasks assigned were basically pointing the students in a direction given a general framework of what to do. Thus the students had to solve a problem they needed to clarify for themselves up front ("What do I want to do?"). There was no single right solution that needed to be achieved (openness) and – as time allowed – the solution could be elaborated as wanted or as possible for the students. Tasks were, for instance, "Design a program that displays your name and animates the letters to interact with the mouse or keyboard!"

This way the students could get familiar with the concepts they had just learned, explore the programming environment, find solutions for their ideas, and implement and test them. The teacher would go around, encourage the students to explore the possibilities, and intervene only if asked or needed. Usually such a working period ended with the end of a lesson. This way those students who wanted to elaborate their work or to extend or modify their programs could continue to do so at home.

### 5.1.5 Presentation and Reflection

Finally the students uploaded their programs to the Scratch webpage and a few programs were presented to the rest of the class at the beginning of the next lesson. Presentation of the work included presentation and discussion of ideas, problems and strategies. If students discovered and applied new concepts in their program they explained them to the rest of the class.

At the end of the course every student was asked to develop his own game, with the only condition being that all of the learned concepts should be applied. The task resulted in a variety of computer games ranging from

- Basic understanding of programming
- Algorithms:
  - Characteristics (finiteness, clarity, feasibility, general validity)
  - What can be solved algorithmically?
- Basic concepts of programming:
  - Sequence
  - Loops
  - Decisions
  - Variables (local/global)
- Input and output of data
- Arithmetic operations and comparison operators
- Representation of algorithms as Scratch blocks
- Object, message, attributes, methods
- Reading and analyzing programs
- Modifying and extending of programs
- Designing, implementing and testing of programs
- Idea generation and problem management

**Figure 1: Educational objectives**

pong and memory to sport and shooting games.

## 5.2 Scratch

The visual programming (mini) language Scratch was originally designed for young students to develop 21<sup>st</sup> century skills (Maloney 2004). It allows creating animations, games and other programs by ‘clicking together’ programming constructs represented as building blocks. Nevertheless, due to its intuitive appearance and usability it is used in computer club houses, high schools and even in introductory programming college courses. We chose Scratch because it emphasizes the practical learning of fundamental CS concepts and at the same time supports the idea of fostering creativity in CS classes. Mini languages are said to provide an insight into programming and teach algorithmic thinking for general computer science in an intuitive, simple, but powerful way (Brusilovsky et al. 1997). Thus Scratch meets the needs for the intended purpose.

## 6 Evaluation

### 6.1 Method

The attempt to introduce programming was done in parallel in two courses of the school. Parallel to the author conducting a lesson as described above (A), the other course (B) was taught by an experienced teacher following a ‘traditional’<sup>5</sup> problem-solving oriented approach. The problem-solving approach was performed by using the tool ‘Robot Karol’ and followed suggested learning tasks as provided with a schoolbook for CS education (Engelmann 2004).

Course A consisted of 21 students, aged 17, with 38% female students. Course B consisted of 23 students of the same age with 61% female students. The students’ prior experience in computer science was comparable.

The accompanying evaluation was following two questions. First, if creativity is explicitly considered, what effect does this have on the students’ motivation, interest, and picture of the school subject of CS? Second, what is the impact on the students’ task understanding and achievement? As a research instrument for the first question, a questionnaire was used; for the second, the average grades of the students before and after the course, and a test following the course.

The questionnaire was structured the following way:

1. Scale-based responses to statements about computer science lessons in general, e.g. “CS lessons are fun/interesting/creative”, “I participate / I am distracted”, “I show results at home”
2. Questions about difficulty, amount of content and appropriateness of the last teaching unit
3. Appraisal of teaching techniques, methods and tools

<sup>5</sup> As a ‘traditional’ problem-solving approach we consider the way a majority of teachers introduce programming by assigning a sequence of convergent problem-solving tasks with increasing difficulty.

4. Scale-based responses to statements about the topic, e.g. “I could discover new things”, “I could concentrate”, “I have the feeling I learned something”
5. Questions about the perceived learning outcome/success of the individual and of the learning group

The questionnaire was answered by the students before and after the 4-week (11-hour) course.

The test following the course contained two sections:

1. Theoretical:
  - Definition and characteristics of algorithms
  - Describing concepts of programming and giving an example
2. Practical:
  - Explaining and optimizing two programs presented on paper
  - Implementing a program to a given problem
  - Implementing a program for a self-chosen task, applying all used concepts

## 6.2 Results

### 6.2.1 Motivation, Interest, Picture of CS

The picture students have of the school subject of computer science is forming their understanding of the science in general. Furthermore this is responsible for students’ motivation and eventually builds the foundation for the question whether they will consider CS as a subject to study at university. Humbert (2003) investigated students’ pictures of CS in his dissertation research. The subject was seen as the science of computers and of how to use computers. The chance of designing and shaping software systems was rarely reflected. This view did not change much after one year of CS lessons.

The creativity-teaching unit changed the students’ picture of CS in many ways, as illustrated in Figure 2, which shows the change in responses between the start and end of the course. Fun and interest were raised considerably (“Computer science is fun” (71% → 93% agreement), “I regard the content of computer science as interesting” (29% → 93% agreement)). These factors have a major impact on the motivation of the students and can be greatly used for maintaining students’ interest in CS. Programming was very motivating for the students – unlike many experiences in the classroom and in

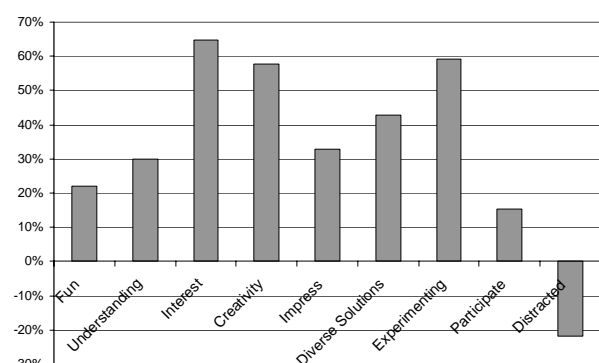


Figure 2: Agreement compared with before the lesson

introductory programming courses at university, where programming is often a reason for failure.

Consistent with the teaching approach, a big change happened in the judgment of creativity. In response to whether they consider CS lessons as a place where they can be creative, 93% answered in the affirmative, compared with 36% before.

Computer science is generally a subject where several solutions are possible for a task and where experimentation is also involved in understanding difficult interrelations. Experimentation is playing an increasingly important role, e.g. for analyzing the behavior of software (Reed 2002). Often in school settings these aspects are not obvious to the students. This is especially true when teachers need to choose 'effective' ways of teaching, such as teacher-centered instruction with convergent problem-solving tasks to 'get through the stuff' in the shortest time. Here, too, the majority of students are not aware of these aspects prior to the creative lessons. Afterwards most agreed that 'in CS diverse solutions and solution methods are possible for a single task' (43% → 86%) and 'in CS lessons you experiment a lot' (14% → 73%). This increase is especially interesting as the students in the previous teaching unit were actually investigating, designing and experimenting with databases. Apparently, genuine designing and changing a computer system by programming in a creative way better meets the students' understanding of 'experimenting' than investigating the characteristics of a 'fixed' system such as a database with convergent problem-solving tasks.

The creation of presentable products (programs) can also have an effect on how students' friends and family consider CS lessons. From almost none at the beginning (7%), 40% of the students agreed that they could impress family or friends with results from the lessons.

Summarizing, the students' picture of computer science lessons changed positively. CS lessons in German schools – and in other parts of the world as well – often differ a lot from real CS. They are perceived as the subject where you learn how to use the computer, how to use Word and Excel and how to use the internet. The students are now more aware of the reality of CS, as a subject that involves designing and changing computer systems, experimenting, and finding a good solution where many solutions are possible. As high school also needs to prepare students for university, these factors need to be considered. Furthermore the 'technical' reputation of CS has caused a gender bias, with girls in particular not showing much interest. With the creative way of looking into CS the girls' interest was also raised and they enjoyed the tasks. Considering the answers separated by gender, it is apparent that the girls mostly answered comparably with the boys.

## 6.2.2 Understanding, Achievement

The perceived learning outcome was stated as high. The answers are in accord throughout the class and stand in contrast to the perceived learning outcome of the previous teaching unit. There the answers are more diverse, and

half of them include reports of problems. These views are also reflected in the perceived learning outcome judgment for the course: 87% believe that all or most of the students in the class understood the programming content well or very well. In the previous teaching unit, the majority of the students checked answers reflecting problems among their classmates. This is interesting, as their own reported learning success in 'databases' was generally better than the perceived learning success of the class. Even if at least half of the students understood the matter, the class was learning in a climate of problems and 'not-understanding'. In the topic of programming, due to the many ways of presenting the students' results and achievements, the classroom climate was a more positive one. This in turn could motivate the students' persistence and desire to understand when they encountered problems.

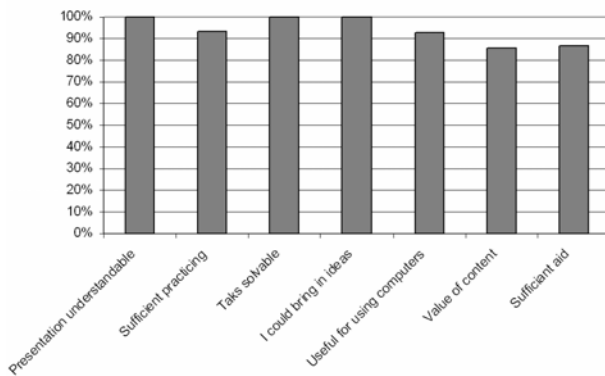
The effective learning outcomes were measured by a test concluding the teaching unit. The test was successfully accomplished by 94% of the participating students. The course average of the test is 0.2 grades better than the class average in the first half of the semester and about one grade better than the average in the test concluding the previous teaching unit. The grades can be separated into two groups: 69% received grades 'good' (2) or 'very good' (1), 25% 'satisfactory' (3) or 'fair' (4). Considering the grades according to gender, all girls received grades of 2 and better, while the boys' grades are equally distributed through the scale. Keeping in mind the problems many CS and programming courses have with female students' achievements, this seems to be an encouraging outcome.

## 6.2.3 Additional Results

### 6.2.3.1 Questions about the lessons

Unfortunately students are generally not used to working independently in the classroom. Even though pedagogy has for decades suggested different teaching methods, the most prominent teaching style in German schools is still teacher-centered classroom instruction (Meyer 2003). Applying a new teaching method can be challenging and troublesome for all participants, as the students may not be sure about what they are expected to do, and cannot follow a common familiar schema. In this connection it is interesting to investigate the perception and the attitude of the students towards the teaching methods. Students' answers about the teaching methods, tasks, and lessons are presented in figure 3.

All students considered the presentation of the learning content as understandable. This is a desired result, but still surprising, for two reasons. First, the teacher did not put much effort into explaining and concretizing the concepts of programming. More or less anything that was learned was done so by actively engaging in programming. Content was presented briefly or collected together and applied right away. This approach is supported by the constructionist learning theory (Papert 1980) that encourages learning by design and engaging students in personal meaningful tasks. Second, programming is known for being a difficult matter to



**Figure 3: Appraisal of teaching methods**

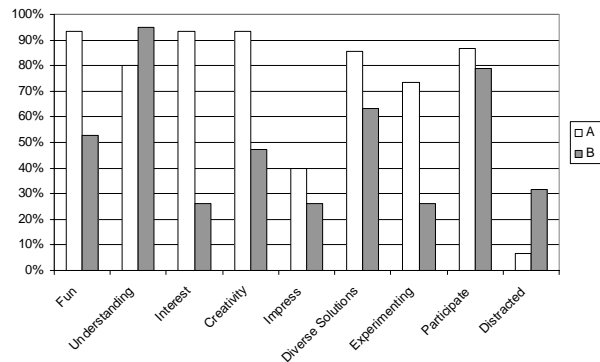
teach in schools (many teachers in high schools struggle for months and even years to teach the basic concepts of programming). The students here considered the degree of difficulty as appropriate and perceived their learning outcome as appropriate or a lot. Asked about how hard the subject matter was, half of the students responded ‘sometimes easy, sometimes hard’, the other half ‘generally easy’.

Practice time was perceived as adequate, even if the circumstances put quite some time pressure on the students. Students appreciated that they could bring their own ideas. All students considered the tasks as solvable – even if there was no ‘right’ solution that they needed to find for the tasks.

The role of the teacher is reflected in the answers to the question about where the students learned most: by working at projects (60%) and dealing with the tasks (60%) in contrast to explanations of the teacher (13%). Again, it is somewhat surprising that programming, at least at this elementary level, can be learned so intuitively.

#### 6.2.3.2 Questions about the topic

The answers about the topic are somewhat ambivalent. Today’s students grow up surrounded by technology. Every student in the observed class has a personal computer at home. Nevertheless less than a third of the students stated that the topic was dealing with issues out of everyday life and only 43% of the students think they can use the learned knowledge in future. These numbers are even lower than the ratings for the previous teaching unit. In the classroom the relevance of programming concepts and the connection to everyday life have not been explicitly shown to the students by the teacher. Given the strength of real-life contexts as a major source of motivation, one would think that the students could hardly have been motivated for the lesson. Surprisingly, 80% stated that some of the issues had been very interesting to them, 87% state they learned something, and 73% said that they had fun with this topic. Obviously – without being motivated by the topic as being connected to everyday life – the tasks and the creative practice were motivating enough for the students to enjoy and learn.



**Figure 4: Agreement compared with control group**

## 6.2.4 Comparison with the control group

### 6.2.4.1 Comparing the questionnaires

Prior to the introduction to programming the curriculum of both courses consisted of the same topics in computer science<sup>6</sup>. The grades of the two courses prior to the observation were generally comparable.

Comparing the answers of the two courses about their picture of computer science prior to the introduction to programming the students answered very much alike. B had slightly more consent with the item ‘fun’ and a significantly higher consent with the item ‘understanding’. The rest of the answers are comparable.

This changes tremendously when comparing the answers after the introduction to programming: fun rose by 22% in A but declined by 32% in B. Only half of the students of B considered CS as fun after the programming course. While programming had an enormous impact on the interest in CS of the students of A, in B ‘interest’ remained low for 75% of the students. Students of group B also agreed more on ‘creativity’ after learning about programming (+18%). This is an interesting fact, showing that even in the problem-solving approach creativity is needed, and this also becomes obvious to the learners. Similar findings are reported by Long (2007). Not all students in the problem-solving group saw that there are several ways of solving a given problem. The agreement with this item rose in B to 63%, while in A it rose to 86%. The agreements to the statements in comparison with the control group are illustrated in Figure 4.

### 6.2.4.2 Comparing the achievements

The course was started in both groups with the same learning objectives in mind. Unfortunately not all learning objectives could be achieved in group B. Variables are not implemented in the tool used in B. Also characteristics of algorithms were not considered by the teacher of group B due to a lack of time. Even so, it is interesting to compare the answers of the students as to how they perceived their achievements.

<sup>6</sup> We will refer to the course with a creative introduction to programming as A and the course following a problem-solving approach as B.

In the questionnaire both groups had to assess their learning success. While in A all students stated that it was appropriate or high, in B only two-thirds of the students did so. Nevertheless the grades of the following test were comparable. Both groups considered the difficulty of the lesson and the complexity similarly (appropriate or high).

Comparing the grades with those of the prior teaching unit, group A improved a lot while group B on average remained stable at the grades they had before. But splitting the grades by gender, the boys of group B improved their grades while the girls deteriorated. In contrast to that stand the achievements of group A, where the girls improved their grades considerably more than the boys. Before the introduction to programming, the level was equal for boys and girls in both groups.

### 6.3 Critical Reflection

There are two drawbacks to this study. First, the lessons were taught by different teachers. The teaching style, the teachers' personalities and the way the teacher gets along with the class can and will have an effect on the learning outcome and the students' motivation. On the other hand, as was shown by the questionnaire that the students completed before the observed lessons, motivation and achievement were equally high in both groups. But as the tasks and exercises used in the problem-solving approach have been taken from a school book that has been used by hundreds of teachers before to introduce programming, they seem to be quite typical for a course that introduces programming through problem solving.

Second, the groups had not only a different methodology but also different software tools. This is perhaps a key factor, and might be responsible for the rise in motivation and perceived creativity as well. Nevertheless, the bottom line stays the same: a creative introduction to programming is both possible and expedient. If the reason is the programming language used, it might be wise to consider creativity when choosing a programming language. As the creativity criteria fit well with many pedagogical implications, they should at least be considered. If the reason for the success of the teaching unit lies in the application of the creativity factors, it is even more strongly recommended that these factors be applied in other teaching settings. Besides, choosing a programming environment that addresses the students' interests will be helpful anyway. Scratch is obviously a candidate for that. We strongly believe that both – the application of creativity and the creativity support of Scratch – are responsible for the learning success. Hence we suggest that creativity be applied to introductory programming courses, regardless of the programming language used, but particularly if using Scratch.

Future research needs to address these questions in detail.

## 7 Conclusion

Analyzing the literature in CS education research we found that creativity is rarely regarded, especially in high school education. Promising results from applying creativity are described by a few authors. We applied a framework for designing creative CS lessons based on a

set of creativity criteria. The conducted teaching unit in introductory programming fulfilled the expectations: the students enjoyed the lessons, the learning objectives were met and the students' picture of CS improved. This is in agreement with studies where contextualization, personalization, and choice produced dramatic increases in students' motivation, in their depth of engagement in learning, in the amount they learned in a fixed time period, and in their perceived competence and levels of aspiration (Cordova and Lepper 1996).

The students' efforts were concentrated and intrinsically motivated. Even when a lesson was over, many of them wanted to remain in the classroom in order to continue working on their project. The presentation and dissemination of the students' results led to increased motivation in the next lesson. Even another course at the school was getting to know the results of this course as many students soon started to play online the games that they had created.

Female students performed very well in the course and could engage in tasks they enjoyed. Our initial impression of a few female students was that they were likely to get distracted by designing the look of the program and less interested in focusing on the functionality; for example, that they would be more interested in making little films than interactive programs. But as soon as some programs were presented, challenged by the creative classroom climate, they caught up and applied the newly learned concepts as well. Especially contrasting the learning results to the control group it becomes obvious that female students performed better in the creative teaching setting.

Interestingly we found that sometimes it is not easy to change a firm stereotype of CS, as illustrated by the following example. After the lessons one student seemed quite unhappy and uncertain. When she was asked about what was bothering her she answered that she found the lessons a bit strange and asked when we would start 'real' CS. The experienced lessons in her opinion had been so "c-r-e-a-t-i-v-e". In her opinion, other subjects are supposed to be creative, but not CS. Asked whether she understood the content and enjoyed the lessons, she said that she had. The lessons just had not met her pre-conceived notion of CS.

After these experiences we believe that creativity can and should be applied in the long run in programming courses and can possibly serve as a principle in other fields of CS as well. We would also like to encourage educators of CS to apply creativity at the university level. The benefits of increased motivation and interest for all students, but especially for women, are worth trying and come with a low risk. Since the beginning CS has been a creative endeavor (Scragg et al. 1994, Saunders 2005). Let us show our students what this can mean.

More and more learning environments are developed that allow a smooth – and creative – introduction to programming. The full potential that lies in these powerful tools can be better tapped with regard to creativity.

In a time where standardized tests are becoming more and more common, the call for more attention to something that is seen as ineffective as creativity may seem a little odd. Nevertheless, the positive outcomes as described encourage us to further investigate how creativity can be fostered, at the same time enhancing learning in computer science education.

## 8 References

- Bergin, S. and Reilly, R. (2005): The Influence of Motivation and Comfort-Level on Learning to Program. In *Proc. of PPIG 17*. University of Sussex, Brighton UK, 293-304.
- Boden, M. A. (1990): *The creative mind: myths & mechanisms*. Basic Books, London.
- Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. (1997): Mini-languages: a way to learn programming principles. *Education and Information Technologies*, **2**, 65-83.
- Chaytor, L. and Leung, S. (2003): How to creatively communicate Microsoft.NET technologies in the IT curriculum. In *Proc. of the 4th conference on Information technology curriculum*, Lafayette, Indiana, USA, 168-173, ACM Press.
- Clements, D. H. (1995): Teaching Creativity with Computers. *Educational Psychology Review*, **7**(2): 141-161.
- Cordova, D. and Lepper, M. (1996): Intrinsic Motivation and the Process of Learning: Beneficial Effects of Contextualization, Personalization, and Choice. *Journal of Educational Psychology*, **88**(4): 715-730.
- Curzon, P. and Rix, J. (1998): Why do Students take Programming Modules? In *Proc. of the 6th annual conference on the teaching and computing and the 3rd annual conference on integrating technology into CSE: Changing the delivery of Computer Science Education. ITICSE '98*. Dublin, Ireland, 59-63.
- Engelmann, L. (2004): *Informatische Grundbildung*, Paetec, Altenburg.
- Epstein, R. G. (2006): An ethics and security course for students in computer science and information technology. In *Proc. of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA, 535-537, ACM Press.
- Fasko, D. (2000): Education and creativity. *Creativity Research Journal*, **13**(3-4): 317-327.
- Feldgen, M. and Clua, O. (2003): New motivations are required for freshman introductory programming. In *Proc. of the 33rd ASSE/IEEE Frontiers in Education Conference*. Boulder, USA, **1**: T3C-T24.
- Gardner, H. (1993): *Creating minds: an anatomy of creativity seen through the lives of Freud, Einstein, Picasso, Stravinsky, Eliot, Graham, and Gandhi*, BasicBooks, New York.
- Glass, R. L. (2006): *Software creativity 2.0*, developer .\*, Books, Atlanta.
- Gu, M. and Tong, X. (2004): Towards Hypotheses on Creativity in Software Development. *Lecture Notes in Computer Science*, **3009**: 47-61.
- Guzdial, M. and Soloway, E. (2002): Teaching the Nintendo generation to program. *Communications. of the ACM*, **45**(4): 17-21.
- Hill, A. M. (1998): Problem solving in real-life contexts: An alternative for design in technology education. *International Journal of Technology and Design Education*, **5**(3), 1-18.
- Hubwieser, P. (2000): *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele*. Springer, Berlin.
- Humbert, L. (2003): *Zur wissenschaftlichen Fundierung der Schulinformatik*, Pad-Verl., Witten.
- Janneck, M. (2006): Partizipative Systementwicklung im Informatikunterricht. *LOG IN* 138/139: 60-66.
- Kaasbøll, J. J. (1998): Teaching critical thinking and problem defining skills. *Education and information Technologies*, **3**(2): 101-117.
- Kaufman, J. C. and Sternberg, R. J. (2007): Creativity. *Change: The Magazine of Higher Learning*, **39**(4): 55-60.
- Lakhani, K. and Wolf, R. (2005): Why Hackers Do What They Do: Understanding Motivation Effort in Free/Open Source Software Projects. In *Perspectives on Free and Open Source Software*. 3-22. J. Feller, B. F., S. Hissam, and K. R. Lakhani (eds). MIT Press.
- Leach, R. J., Ayers, Caprice A. (2005): The Psychology of Invention in Computer Science. In *Proc. of 17th Annual Workshop of the PPIG*. University of Sussex, Brighton UK.
- Lewandowski, G., Johnson, E. and Goldweber, M. (2005): Fostering a Creative Interest in Computer Science. In *Proc. of SIGCSE '05*. St. Louis, MO.
- Lewis, T., Petrina, S. and Hile, A. M. (1998): Problem Posing-Adding a Creative Increment to Technological Problem Solving. *Journal of Industrial Teacher Education*, **36**(1).
- Long, J. (2007): Just For Fun: Using Programming Games in Software Programming Training and Education - A Field Study of IBM Robocode Community. *Journal of Information Technology Education*, **6**: 279-290.
- Maloney, B., Kafai, Rusk, Silverman, Resnick (2004): Scratch: A Sneak Preview. *IEEE Computer Society*, 104 - 109.
- Mamone, S. (1992): Empirical Study of Motivation in an Entry Level Programming Course. *ACM SIGPLAN Notices*, **27**(3): 54-60.
- Meisalo, V., Sutinen, E. and Tarhio, J. (1997): CLAP: teaching data structures in a creative way. In *Proc. of the 2nd conference on Integrating technology into computer science education*. Uppsala, Sweden, 117-119.
- Meyer, H. (2003): *Unterrichtsmethoden II: Praxisband*, Cornelsen Scriptor, Berlin.
- Mittermeir, R. (2000): Informatik-Unterricht: Bastel-Unterricht, eine intellektuelle Herausforderung oder "Preparation for the information-age". *Medienimpulse*, **9/33**, 4 - 11.

- Papert, S. (1980): *Mindstorms : children, computers, and powerful ideas*, Basic Books, New York.
- Peppler, K. and Kafai, Y. (2005): Creative Coding: Programming for Personal Expression. <http://scratch.mit.edu/files/CreativeCoding.pdf>. Accessed 19 Oct 2007.
- Reed, D. (2002): The use of ill-defined problems for developing problem-solving and empirical skills in CS1 *J. Comput. Small Coll.* **18**(1): 121-133.
- Resnick (2002): Rethinking Learning in the Digital Age. In *The Global Information Technology Report: Readiness for the Networked World*. 32-37. Kirkman, G. (ed). Oxford University Press, Oxford.
- Resnick, M. (2007): All I really need to know (about creative thinking) I learned (by studying how children learn) in kindergarten. In *Proc. of the 6th ACM SIGCHI conference on Creativity & cognition*, Washington, DC, USA, 1-6, ACM Press.
- Rich, L., Perry, H. and Guzdial, M. (2004): A CS1 course designed to address interests of women. In *Proc. of the 35th SIGCSE technical symposium on Computer science education*, Norfolk, Virginia, USA, 190-194, ACM Press.
- Romeike, R. (2006): Creative students - What can we learn from them for teaching computer science, In A. Berglund & M. Wiggberg (Eds.) In *Proc. of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling*. Uppsala University, Uppsala, Sweden. Also available at <http://cs.joensuu.fi/kolistelut/>
- Romeike, R. (2007a): Kriterien kreativen Informatikunterrichts. In *Proc. of the 12. GI-Fachtagung "Informatik und Schule - INFOS 2007"*. Siegen, Germany, LNI 112: 57-68. Köllen.
- Romeike, R. (2007b): *Designing Animations and Games - A Creative Introduction to Programming: About flying Elephants, Dogs, Cats and Ideas!* <http://www.funlearning.de/>. Accessed 19 Oct 2007.
- Romeike, R. (2007c): Three Drivers for Creativity in Computer Science Education. In *Proc. of the IFIP-Conference on "Informatics, Mathematics and ICT: a golden triangle"*. Boston, USA.
- Romeike, R. and Schwill, A. (2006): "The studies might be too difficult for me" Intermediate Results of a Long-term Survey of Computer Science Freshmen. In *Proc. of HDI 2006: Hochschuldidaktik der Informatik*. Munich, P-100: 37-49, Lecture Notes in Informatics.
- Runco, M. A. and Chand, I. (1995): Cognition and Creativity. *Educational Psychology Review*, **7**(3): 243-267.
- Saunders, D. and Thagard, P. (2005): Creativity in Computer Science. In *Creativity across domains: Faces of the muse* (Ed, Baer, J. C. K. a. J.), Lawrence Erlbaum Associates, Mahwah, NJ.
- Scragg, G., Baldwin, D. and Koomen, H. (1994): Computer science needs an insight-based curriculum. In *Proc. of the twenty-fifth SIGCSE symposium on Computer science education*, Phoenix, Arizona, United States, 150-154, ACM Press.
- Shneiderman, B. (2000): Creating Creativity: User Interfaces for Supporting Innovation. *ACM Transactions on Computer-Human Interaction*, **7**(1): 114-138.
- Sternberg, R. J. and Lubart, T. I. (1991): Creating Creative Minds. *Phi Delta Kappan*, **72**, 608-614.
- Sutinen, E. and Tarhio, J. (2001): Teaching to identify problems in a creative way. In *Proc. of the Frontiers in Education Conference*, IEEE Computer Society, TID-8-TID-13vol.1.
- Sweeney, R. B. (2003): Creativity in the Information Technology Curriculum Proposal. In *Proc. of the 4th conference on Information technology curriculum*. Lafayette, Indiana, USA, 139-141.
- Tharp, A. L. (1981): Getting more oomph from programming exercises. *SIGCSE Bull.*, **13**(1): 91-95.
- Thomas, J. C., Lee, A. and Danis, C. (2002): Enhancing Creative Design via Software Tools. *Communications of the ACM*, **45**, 112 - 115.
- Van Dyke, C. (1987): Taking "computer literacy" literally. *Communications of the ACM*, **30**, 366-374.
- Westby, E. L. and Dawson, V. L. (1995): Creativity: Asset or Burden in the Classroom? *Creativity Research Journal*, **8**(1): 1-10.
- Wilson, B. C. (2004): A study of learning environments associated with computer courses: can we teach them better? **20**(2): 267 - 273.

# Student Transformative Learning in Software Engineering and Design: discontinuity (pre)serves meaning

Leslie Schwartzman

Department of Computer Science  
Roosevelt University  
Chicago, Illinois USA

sla@acm.org

## Abstract

Reflective practice is considered to play an important role in transformative learning of educationally critical material, but students often respond in other, less productive ways. Transformative learning is used here as a lens to investigate reflectiveness: understanding the place of reflectiveness – and how defensiveness has no place – in transformative learning illuminates its operation and mechanism. The paper is written as part of an ongoing exploration into how to engender students' reflective response to difficult material, preparing a foundation on which to address that question directly. Previous preparation includes phenomenological analysis of reflectiveness and defensiveness, and a careful examination of the operation and mechanism of defensiveness, both based on Segal's explication of Heidegger's dynamic of rupture. Qualitative data for the investigation comes from an upper-level undergraduate software engineering and design course that students invariably find quite challenging. A grasp of concepts presented here should enable faculty to develop improved pedagogy and institutions to design more effective curricula for engendering students' reflective response to difficult material in computing – and other – education.

**Keywords:** defensiveness, reflectiveness, phenomenology, experiential learning, dynamic of rupture, pedagogy, curriculum, student feedback, confusion

## 1 Introduction

*The real voyage of discovery consists not in seeing new landscapes but in having new eyes.* – Marcel Proust

True learning begins as an encounter with the unknown and existentially unfamiliar, and therefore includes some interval of confusion that the student must navigate productively in order to reach knowing. Alternatively, if the student does not navigate that interval productively, an encounter with the unknown and existentially unfamiliar leads not to learning but to other, more problematic outcomes. In the former, the student is said

to respond reflectively; in the latter, the student is said to respond defensively. This paper is written as part of an ongoing exploration into the operation and mechanisms involved in students' responding reflectively (to navigate the interval of confusion productively), and what is required to support it.

Preparation is required before addressing that question directly. The exploration began with a phenomenological investigation of reflectiveness and defensiveness, based on the literature. In particular, Segal's explication of Heidegger's dynamic of rupture provides a conceptual structure for analyzing students' experience: the sequence *rupture* → *explicitness* → *response* (either *reflective* or *defensive*) (Segal 1999). On that basis, the first paper contained careful definitions of reflectiveness and defensiveness, with some guidelines on recognizing instances of the dynamic of rupture among students and discriminating between the two possible responses (Schwartzman 2006). A subsequent paper analyzed the treatment of defensiveness by several classic sources (Segal alone – oriented in the gestalt of the dynamic – regards it as equally substantive with reflectiveness), and delineated more precisely its operation and mechanisms (Schwartzman 2007).

Qualitative data for the investigation comes from an upper-level undergraduate software engineering and design course that students invariably find quite challenging.

### 1.1 The Research Question

The ongoing exploration is motivated by the question most educationally productive, relevant to reflectiveness: how to engender a reflective response (which is required for transformative learning) among all students; or, among those students who do respond defensively, how to cultivate transition to reflectiveness. Building on considerations already addressed, the exploration continues here with an investigation into reflectiveness, viewed through the lens of transformative learning, to delineate more precisely its operation and mechanisms.

### 1.2 Pedagogy and Curriculum

An established profession or discipline is characterized by a body of esoteric knowledge that a prospective practitioner must master in order to become – and be recognized by the community as – a skilled professional (Johnson 2001). That esoteric knowledge includes difficult, counter-intuitive concepts so critical to

---

Copyright © 2008, Australian Computer Society, Inc. This paper appeared at 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 (CRPIT), Vol. 88. Raymond Lister and Simon, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

understanding the discipline that they permanently transform the practitioner's view of it. Making the transition from not-knowing to knowing that esoteric content, successfully navigating passage through the interval of confusion, depends on transformative learning. Such learning does not happen in a cumulative or linear fashion; it is often little understood and poorly supported in the formal education and training into a discipline. Understanding the role of reflectiveness (and the absence of defensiveness) in transformative learning should enable faculty to develop improved pedagogy and institutions to design more effective curricula for engendering students' reflective responses to difficult material in computing – and other – education

### 1.3 Organization of the Paper

Section 2 summarizes the literature background. Section 3 introduces transformative learning. Section 4 speaks to the role of reflectiveness in transformative learning. Section 5 holds a brief overview of software design course CSX, source of qualitative data for this paper. Section 6 contains data. Section 7 contains discussion of implications of this work. Section 8 holds conclusions and possibilities for future work.

## 2 Background Literature

This section contains a brief overview of reflectiveness and defensiveness, and an introduction to some classic literature sources on transformative learning.

### 2.1 The Dynamic of Rupture

Segal's analysis of reflectiveness and defensiveness uses Heidegger's dynamic of rupture as a conceptual framework (Segal 1999, citing Heidegger (1985)). For more information, see (Schwartzman 2007).

#### 2.1.1 Overview

This dynamic can be explained by Segal's (1999) example from Dreyfus (1993), familiar to anyone who is traveling internationally for the first time, perhaps to attend a conference: Each of us 'knows' what particular distance to stand apart from an acquaintance when engaged in conversation. In general, we have no awareness of the specific distance, or even that we are doing it. This 'know-how' resides in the realm of the unseen taken-for-granted. However, when we encounter conference host country natives who use a different conversational distance, we experience them as standing uncomfortably close or uncomfortably far away, and we suddenly become aware that we have an accustomed distance. According to Segal – and it is borne out by the student data from CSX – this discomfort is experienced either productively (reflectively) or unproductively (defensively). Segal's explanation of distinct forms of differentness clarifies the two possibilities.

#### 2.1.2 Distinct Forms of Differentness

Segal, citing Bauman (1990), distinguishes between two kinds of differentness or otherness: the oppositional (in

shorthand, *enemy*) and the unknown (in shorthand, *stranger*). The oppositional is defined according to the same rules as we, but oppositely. Continuing the example of interpersonal conversational distance, the international traveler may respond: "These unrefined (host country) natives are standing the wrong distance away. I can't possibly carry on a civilized conversation under such conditions." Their differentness is thus defined in opposition: their 'wrong' vs. one's own 'correct' distance, 'unrefined' vs. 'refined' nature, 'uncivilized' vs. 'civilized' actions. Defining the other in opposition, as *enemy*, confirms one's view of the world. Questioning of one's own or the other's behavior has no place. *Enemies* oppose each other but have a common appreciation of the terms on which they meet; they *function in the space of the existentially familiar* ...

Alternatively, the unknown is defined by unknown rules, or perhaps not defined at all. The international traveler may respond, "What is happening here?", and eventually, "What does this mean? Do I have an accustomed distance? If so, how did I learn it, what length is it measured at? Do they have an accustomed distance? If so, how do I learn it, what length is it measured at, and how do I figure it out? How long will it take to learn, what will I do in the meantime? ..." Recognizing the other as unknown, as *stranger*, evinces the inadequacy of our worldview. Questions, but no real answers, abound.

*Strangers* have no common understanding of the terms on which they meet. *[They] give rise to the existentially unfamiliar ... [T]here are no ways of reading [such] a situation that can be taken for granted. ... The anxiety of strangeness is experienced not only in the face of the stranger but in the face of strange and unfamiliar situations – in any situation in which we cannot assume our familiar ways of doing things* (Segal 1999 p.76, citing and quoting Bauman (1990 pp.143-145)).

#### 2.1.3 Reflectiveness and Defensiveness

In summary, rupture is required for explicitness; explicitness serves as pre-condition to both reflectiveness and defensiveness, which include significant affective components: unease and uncertainty are engendered by the shock of estrangement following rupture and explicitness. Defensiveness shields the responder from having to experience the estrangement, the unease, and the uncertainty: one disassociates from engendered uncertainty by recasting the unknown (strange) explicit as the known oppositional (enemy) explicit; one disowns engendered unease by projecting or displacing all responsibility for difficulty onto that recast source. In defensive response, one avoids the challenges of uncertainty and its affective components; for example, the first possible response attributed to the international traveler (2.1.2). In reflective response, one takes on those challenges; for example, the second possible response attributed to the traveler (2.1.2).

## 2.2 Transformative Learning

This paper draws from several classic literature sources on transformative learning, all speaking virtually of one

accord about reflectiveness, albeit from varying perspectives. Each organizes the information in a distinct way, using different vocabularies: *How We Think*, John Dewey's 1910 book on educating for reflective thought; *The Structure of Scientific Revolutions*, Thomas Kuhn's seminal book on the history and historiography of science; and *Transformative Dimensions of Adult Learning*, Jack Mezirow's study of adults' transformative learning. Dewey is interested in developing students' capacity for critical thought so that it becomes available for application to any content domain. He uses the study of various academic subjects as specific examples for the general habit of mind he wishes to inculcate: well-prepared reflectiveness (Dewey 1991). As a historian of science, Kuhn studies patterns of 'learning process' in communities of practice. He directs attention to the nature of *discovery*, whereby principles are reformulated and practice transformed. The process begins in anomaly, conflict between observed phenomena and a community's operative model for interpreting the world. Kuhn examines conditions under which that conflict gives rise to new models, tension generated by competition among models for dominance, and ways in which scientific communities navigate and ultimately resolve that tension (Kuhn 1996). Mezirow chaired the Department of Higher and Adult Education at Columbia University. In addition to reporting his own research, he draws on and cites a wide spectrum of scholarship related to transformation theory (Mezirow 1991).

The work of Aron Gurwitsch (1964) on the organization of fields of consciousness provides a rich framework for discussing various concepts relevant to the investigation.

Elements of striking symmetry connect Kuhn's work on the development of new scientific knowledge with that of Dewey on an individual's intellectual progress. The discovery process of an established research community encountering persistent anomalies appears almost as an isomorphic projection from the learning process of an individual encountering educationally critical, counter-intuitive concepts in an established body of knowledge.

### 3 About Transformative Learning

Introductions to components, a situating context, and well understood aspects, as well as definitions, follow.

#### 3.1 Components of Transformative Learning

##### 3.1.1 Meaning

Every source cited for this paper talks explicitly about the primacy of meaning. Mezirow defines meaning as *an interpretation, and to make meaning is to construe or interpret experience ... [done] both prelinguistically ... and through language ... [by] processes involving awareness. In other words, it gives coherence to experience* (Mezirow 1991 p.4). Dewey considers the absence of meaning an anomaly; he believes that the exercise of intelligence requires its existence, and to grasp it constitutes *the nerves of our intellectual life* (Dewey 1991 p.116). Meaning amounts to a coherent representation of experience, but cannot be arbitrarily

imposed; it arises out of experience. *Saliency of a group of data so that this group emerges and segregates itself from the stream [of experience] is a feature not introduced into the stream, but yielded by the stream itself* (Gurwitsch 1964 p.31, citing James (1890)).

For Dewey, individual learning may be defined as making meaning; what one can interpret effectively, one understands both differentiated from and in relationship with its surrounding context (Dewey 1991 p.117).

##### 3.1.2 Meaning Frames

Meaning making takes place under an orienting frame of reference, a *structure of assumptions within which one's past experience assimilates and transforms new experience, ... a habitual set of expectations*. Such structures embody the categories and rules that order new experience, shaping how we classify our encounters with the world: what we take in and how we act. They also dictate what we notice and what we ignore by *selectively determin[ing] the scope of our attention ... informed by an horizon of possibility, ... to simplify, organize, and delete what is not salient in sensory input ... and provide the basis for reducing complex inferential tasks to simple judgments*. Thus, they function as both *lions at the gate of awareness and the building blocks of cognition* (Mezirow 1991 pp.49,50).

Further, these structures define composite and prime elements (Kuhn 1996 p.129). A *composite* element can be decomposed to yield component parts relevant to meaning, a *prime* element cannot. For the reader seeking to understand an essay, its sentences (and their decomposition into constituent words) exemplify the former; constituent letters of those words exemplify the latter. For the calligrapher, in contrast, alphabetic letters (and their decomposition into keystrokes) represent composite elements (Simon 1996).

Note that sources for this paper use a variety of terms for these structures: *meaning schemes, meaning perspectives, frameworks of understanding, unconscious principles and assumptions, paradigms, schemas*, etcetera. Meaning frame is here used interchangeably with all of them.

Dewey states that *[e]xplicit thinking goes on within the limits of what is implied or understood*, and describes the role of these 'premises', the grounds or foundations, in reasoning: the premises contain the conclusions and the conclusions contain the premises. The importance of coherence as an organizing principle is embodied in that relationship (Dewey 1991 pp.81,215). Similarly, Kuhn describes operation under a paradigm as a fundamental principle of science, with both the structure and the constraints that imposes: *All research takes place within the context of a paradigm ... All observations are paradigm-based ... The commitments that govern normal science specify not only what sorts of entities the universe does contain, but also, by implication, those that it does not* (Kuhn 1996 pp.79,126).

Meaning frames operate below the level of awareness, as an *unconscious* system of ideas. They inhabit the realm of the unseen, taken-for-granted: *The old, the near, the*

*accustomed, is not that to which but that with which we attend* (Dewey 1991 p.222).

### 3.1.3 Awareness: organization of consciousness

Meaning frames refer to what one brings within oneself to engage in and interpret an encounter with the world. The work of Aron Gurwitsch enables reference to those aspects of the world that one takes in or one acts upon through the meaning frame. He asserts that every field of consciousness, regardless of content, exhibits a *universal, formal pattern of organization* comprising three domains or dimensions: the *thematic focus* or *theme*, upon which one's mental activity concentrates; the *thematic field*, aspects of the world co-present with the thematic focus and having relevance to it; and the *margin*, aspects of the world co-present with the thematic focus but irrelevant to it (Gurwitsch 1964 p.56).

Meaning frame operation and field of consciousness organization exist interdependently: meaning frame dictates what aspects of the world one's awareness encompasses at any given moment, which elements among them have direct significance (corresponding to what occupies the thematic focus), which have significance by association (corresponding to content of the thematic field), and which have little or no significance (corresponding to marginalia).

Booth summarizes Gurwitsch's work: The structure of awareness may be thought of as a dynamic relationship between oneself and the object of one's consciousness. One brings the totality of one's experience and awareness to the perception or consideration of some aspect of that object. The object is said to 'present' itself in that awareness; how it does so determines the thematic focus to emerge from it, and the attendant elements of relevance constituting the thematic field. A shift in one's awareness to another aspect of the object brings a corresponding shift in the thematic focus and thematic field. In contemplation of an object, one shifts one's awareness alternately among its different aspects. Each attendant shifting organization of the dimensions of one's consciousness may be enacted by delineation of a different element of the thematic field into thematic focus (Booth 1997 pp.141,146).

The more differentiated one's view of an object (the more aspects one can bring awareness to), the richer the set of elements in the thematic field associated with it and the more varied the set of elements that can serve as a thematic focus during contemplation of the object; thus, the deeper one's understanding. Consequently, a sparsely populated – or empty – thematic field leaves little possibility for deep understanding.

In the example of the traveler at the conference, conversation with host country natives was taken as the object of her consciousness, and standing distance between her and them became the thematic focus. Elements in the thematic field included observations and recollections. She observed host country natives talking with each other, host country natives talking with attendees from other countries, and attendees from other countries talking among themselves. She recalled

standing distance in conversations at home with close friends or family, or with strangers, and how that distance varied depending on conversational content and the nature of the encounter.

## 3.2 An Embedding Context: learning

Transformative learning is situated within a larger context of learning generally.

### 3.2.1 Experiential Learning

All the classic sources on which this paper draws hold to a model of experiential learning. Sources speaking to it explicitly describe common underlying phenomena and a common process, although they may bring attention to different aspects. Those not speaking to it explicitly rely on the same phenomena and process as implicit model. According to the model, *learning is always grounded in prior experience, and any attempt to promote new learning must take into account that experience* (Boud 2001 p.11). Dewey thinks that, most often, *ordering of thought is attained through ordering of action*, and that tacit knowledge (required for acting effectively) precedes explicit knowledge (required for describing coherently): *[T]he development of an unconscious logical attitude and habit must come before [the conscious use of such an attitude]. ... [The conscious setting forth] is valuable only when a review of the method that achieved success in a given case will throw light upon a new, similar case* (Dewey 1991 p.41,132,113).

### 3.2.2 Meaning Frames: dynamic entities

In the normal course of our encounters with the world, meaning frames undergo endless refinement. Popper states: *We have been born with the task of developing a realistic set of expectations about the world based on the coded messages we receive from it. We can't even be sure of the code but must keep checking [constantly] on it.* (Berkson and Wettersten 1984 p.16). One's *selective and conceptualizing faculties are persistently at work* (Gurwitsch 1964 p.30). The *concept is always under construction* (Kuhn 1996 p.2). Encounters with the world also occur outside that normal course. As Dewey observes, any aspect of the world, no matter how well known, may suddenly present an unexpected and incomprehensible problem (Dewey 1991 p.120). When that occurs, refining the meaning frame does not suffice. Instead, a different kind of learning is required.

### 3.2.3 A Template for Learning

The factors enumerated above characterize CSX students' learning as well. The data illustrate that students learn the material – whether one homework's lesson or the overarching course objectives – only through experience (reinforced by discussion). For that reason, CSX homework assignments are structured to lead students through a series of experiences that bring their practice and (mis)understanding into awareness. Booth endorses this approach (Booth 1997 p.149).

Further, when new experience conforms to expectations grounded in students' extant frameworks of

understanding, learning happens cumulatively and with little additional thought. However, new experience violating expectations creates discontinuity; it leads either to a different kind of learning (when the student responds reflectively) – or to no learning (when the student responds defensively).

### 3.3 Initiation and Consequences

The classic sources concur in their comprehension and descriptions of these aspects of transformative learning.

#### 3.3.1 In the Beginning: discontinuity

Discontinuity in knowing arises from a *conflict between what is known and what must be understood* (Mezirow 1991 p.163, citing Loder (1981)). This conflict, *anomaly*, is defined entirely in relation to one's meaning frame; it occurs as violation of the expectations carried therein. In the absence of an extant meaning frame, anomalies do not exist, by definition. A meaning frame is required to bring the anomalous nature of a phenomenon to light, but inadequate to resolve the problems raised by its existence (Kuhn 1996 p.122). In the example of the traveler, her meaning frame for conversational standing carried the habitual distance to which she'd been socialized; the phenomenon of host country natives' standing distance was defined as an anomaly in relation to it. *Troublesome knowledge* denotes an anomaly that cannot be avoided or ignored by the individual learner (corresponding to what Segal calls *rupture*). *Crisis* denotes the state induced in the relevant scientific community that cannot make an anomaly conform, avoid it, or ignore it (Kuhn 1996 p.ix).

Not all anomalies rise to a level of troublesome knowledge or crisis induction; at a minimum, persistence and significance are required. In the example of the international traveler, if the occasional host country native stands at an unaccustomed distance, the traveler can attribute it to that individual's eccentricities. But if virtually every host country native does so, the anomaly becomes an unavoidable phenomenon whose existential strangeness cannot be ignored.

#### 3.3.2 Aftermath of a new meaning frame

Dewey describes the consequence of reformulation as clarity, illuminating *relations of interdependence between considerations previously unorganized and disconnected ... and binding isolated items into a coherent single whole* (Dewey 1991 p.80). With a new meaning frame, one is re-oriented in the world: the same collection of experience, organized along different principles, embodies a radically different set of relationships. Correspondingly, within a scientific community, *[a]lthough the world does not change with a change of paradigm, the scientist afterward works in a different world* (Kuhn 1996 p.121).

### 3.4 Definition(s)

Transformative learning arises from rupture in knowing. It is here defined in two ways. 1) Directly: through reformulation of meaning frame, transformative learning preserves meaning to effect coherence in transition across

a discontinuity in knowing. 2) Indirectly: transformative learning is both differentiated from and related to deep learning, which occurs cumulatively, as described below.

#### 3.4.1 Differentiation

At the level of individual intellectual progress, deep (cumulative) and transformative learning can be distinguished conceptually using Gurwitsch's fields of consciousness. As a result of deep learning, one switches dynamically – within the same field of consciousness – among thematic foci, with correspondent restructuring of thematic fields. The total set of elements in the field remains constant, while boundaries among the thematic focus, the thematic field, and the margin become fluid, and component elements shift between adjacent domains (Booth 1997 p.144). This corresponds to refinement and clarification of the extant meaning frame.

As a result of transformative learning, the field of consciousness changes: elements formerly not found in any domain of consciousness, possibly including component parts of elements formerly classified as prime, now occupy the thematic focus or reside in the thematic field, and some elements formerly found there are now relegated to the margin. This corresponds to formulation of a new meaning frame.

Kuhn makes an analogous distinction at the level of scientific development: In *normal science*, progress occurs cumulatively, an outgrowth of community practice dictated by and conforming to the reigning paradigm. A new theory (paradigm) doesn't originate via incremental change under these conditions. It begins with anomaly that rises to the level of crisis. *Extraordinary science*, the practice whereby crisis is resolved, requires that prior theory be reconstructed and prior fact be re-evaluated, an *intrinsically revolutionary process*. A new way of sense-making emerges: a way to organize, and then interpret and explain the world. This new paradigm in turn designates what constitutes the significant and what constitutes the irrelevant (Kuhn 1996 pp.7,76,128).

#### 3.4.2 In Relation: cyclical alternation

The classic sources from which this paper draws all describe (albeit using different vocabulary) individual intellectual progress as an alternating rhythm of continuity and discontinuity, of consciousness (which *gives conviction and control*) and unconsciousness (which *gives spontaneity and freshness*) (Dewey 1991 p.217); that is, between deep and transformative learning. Similarly, scientific progress proceeds as alternation of *normal science* and *scientific revolution* (Kuhn 1996).

### 4 Transformative Learning and Reflectiveness

Next, I examine the transformative learning process.

**Note:** 'operation' of a phenomenon refers to what occurs, 'mechanism' refers to how it occurs.

#### 4.1 Transformative Learning: operational view

The classic sources give almost identical accounts of the operation and mechanism of transformative learning.

### 4.1.1 Individual Intellectual Progress

Dewey enumerates *five logically distinct steps common to all types of abstract thinking*: i) a felt difficulty; ii) its location and definition; iii) suggestion of possible solution; iv) development by reasoning of the bearings of the suggestion; and v) further observation and experiment leading to its acceptance or rejection; that is, the conclusion of belief or disbelief (Dewey 1991 p.72).

Barer-Stein (and others cited by Mezirow) developed a five-phase sequence description, a phenomenological analysis of learning as a process of experiencing the unfamiliar: first, *being aware ... [characterized by the dominant question,] What is this?*; second, *observing ... How does this compare with what I know?*; third, *acting ... Shall I try it?*; fourth, *confronting ... Do I know this?*, and *Do I want to?*; fifth (if phase four questions are answered in the affirmative), *involving ... How did this come to be?*, *What are the possibilities, and which makes sense?*, *What [meaning] is relevant for me?* (Mezirow 1991 pp.84-85, quoting Barer-Stein (1987)).

### 4.1.2 Scientific (Community) Progress

Through the course of his book, Kuhn lays out the steps in the development of a *scientific revolution*. 1) *crisis*: observations of persistent anomaly, phenomena that violate the current paradigm with implications that cannot be ignored, and interrupt the practice of normal science (wherein the operative paradigm remains in the realm of the unseen, taken-for-granted); 2) *isolating the difficulty*: determining when, where, and what occurs, and precisely how it violates the current paradigm; 3) *proposal for a new paradigm*; 4) advocates who develop a *strongly reasoned argument in support of the new paradigm*; 5) ongoing debate and experimentation, enacting (for a successful paradigm) *rigorous evaluation, leading to gradual conversion of the relevant scientific community* (Kuhn 1996 pp.152,158).

### 4.1.3 Mechanism: the role of reflectiveness

Sequences enumerated in 4.1.1 define the operation of transformative learning. Reflectiveness is designated as *taking on the challenge of uncertainty and its affective components* (2.1.3), corresponding to steps 2, 3, and 4 in the sequence. That is, reflectiveness serves as the mechanism of transformative learning, and transformative learning is effected as a manifestation of reflectiveness.

To summarize the operational sequence: *discontinuity* → *reflectiveness* → *new meaning frame*. From CSX students' perspectives, it might be more accurately phrased as *confusion* → *struggle* → *knowing*.

## 4.2 Reflectiveness

From the source descriptions in section 4.1, one might construe reflectiveness (in the course of transformative learning) as an intentional process, with each step deliberately chosen in succession. However, as all the sources note, it happens as something other than either conscious choice or linear progression, and is *terminated by ... a relatively sudden ... unstructured event like the*

*gestalt switch* (Kuhn 1996 p.122). Next, I examine both the operation and the mechanisms of reflectiveness.

### 4.2.1 Operational Sequence: the known

Segal, Dewey, and Mezirow (among others) describe the operation of reflectiveness as follows: When anomaly rises to a level of troublesome knowledge, it results in the need to re-evaluate one's meaning frame. One must determine the particular nature of the frame's inadequacies and reformulate it to correct them. Such re-evaluation begins with bringing the frame into one's conscious awareness. The incorporation of heretofore unseen, taken-for-granted elements into the thematic focus and thematic field(s) of one's consciousness is accompanied by much uncertainty and unease. If one can tolerate uncertainty and suspend judgment for long enough, one suddenly finds oneself encountering the world through a new meaning frame.

Using the example of the traveler: prior to the conference, her habitual standing distance resided in the realm of the taken-for-granted unseen. Early in the conference, the host country natives' unaccustomed standing distance could not be ignored or avoided. When she responded reflectively, their standing distance became an element of significance, perhaps intermittently even occupying the thematic focus of her consciousness as she attempted to manage the major distraction it posed. Her expectations were likely revised as follows: "I have been socialized to use a set of conversational standing distances particular to my culture. People in other cultures are socialized to the set of distances particular to their respective cultures. In any encounter with others, I can include within the field of my attention an awareness of our standing distance, adjusting it if necessary, for as long as it takes for us each to feel at ease." By conference end, she would be adjusting her standing distance without deliberate effort to accommodate the various culturally-related patterns – including her own – that she encounters.

### 4.2.2 Mechanism: value in the unknown

Under scrutiny, source descriptions are seen to leave much unexplained and to rely on ambiguous reasoning for describing reflectiveness. For example, Dewey simply states that because reflection originates in a problem, one must *at some points* consciously examine one's implicit assumptions (Dewey 1991 p.215). Further, details of the mechanism effecting reflectiveness, for example, the nature of interplay between conscious and unconscious forces involved, are left unspecified. This vagueness is well-founded; the mechanism of reflectiveness is not well understood, and remains *perhaps ... permanently inscrutable* (Kuhn 1996 p.90).

However, the literature has value; it assists in delineating more precisely what is known and what remains unclear about reflectiveness. In addition, the source descriptions provide a definition in progress for reflectiveness, and a way to approach and discuss what is not well understood.

## 5 CSX: the course, briefly

CSX, an upper-level undergraduate software engineering and design course, provides qualitative data for the paper. For more information, see Schwartzman (2006).

CSX is motivated by concerns about software quality as an ethical issue, and much of the course content is based on the work of David Parnas (2001). It is meant to teach software development fundamentals in a way that transcends software tools and languages, yet engages students in the actual practice of software, not just a theoretical or anecdotal exposition. Every one of the 15 class meetings during a semester includes substantive discussion on several aspects of software development other than code; translation to code (when mentioned) is treated as a small – and the easiest – step in the process. The first 12 of 13 assignments are to be done using pseudocode (for procedures) or English text (for documentation used as a design medium). A group project (assignments 10-13) is begun in class meeting 10. Assignments 10-12 are devoted to using documentation as a design medium, for both functionality and implementation; one half of assignment 12 requires pseudocode. Until assignment 13 (requiring use of a C++ compiler), students are strictly enjoined from coding.

## 6 Qualitative Data

### 6.1 Data Sources

Data for this paper derive from student project logs and end-of-term interviews. During the group project, students are assigned to keep logs (instituted for student accountability), a quantitative record of communication among group members, including participants, dates, times, and tasks accomplished. When students speak in (or outside) class of issues they're wrestling with or thinking deeply about, I invite them to record the material in their logs, promoting their more clarified thought and my more informed teaching. End-of-term interviews were devised for two purposes: they reveal the degree of students' overall knowledge of course material, enabling assessment of individual contribution to a group project; and they provide information about students' learning (or not) process. Initially, for the former purpose, I kept only occasional notes. I began recording interviews (by hand) for later analysis to serve the latter purpose: to better understand – and teach to improve – students' learning experience in the course. Sources are noted.

### 6.2 Methodology and Organization

#### 6.2.1 Introduction to Methodology

Transformative learning occurs as a manifestation of reflectiveness. In this paper, I investigate the former – wherein meaning frames are reformulated – in order to better understand the operation and mechanism of the latter. The nature of a meaning frame is revealed through its operation; the encapsulated unconscious premises and expectations determine how one engages in and interprets encounters with the world. By definition, extant meaning frames elude conscious access, so they cannot be used

directly to study student transformative learning. À la Proust's metaphor (*The real voyage of discovery consists ... in having new eyes*): we cannot *see* our eyes (no one spontaneously speaks about the operation of her meaning frame), we can only see *through* our eyes. One does not have a point of view on one's point of view (Sartre 1966). Instead, internal change is projected onto the world, as indicated by CSX feedback data. Students' statements found in 6.3 make clear they know that – and somewhat how – (their relationship to) relevant aspects of the world have radically changed.

Transformative learning brings into awareness one's former (and formerly taken-for-granted, unseen) meaning frame, experienced – and described – as perceptions of (one's relationship to) relevant aspects of the world. End-of-term accounts often include some variant on the statement "Before, I only knew or did ... as a way to write code; now I know a process that ... as a way to develop software." These accounts present information about transformative learning (or not) by describing – and contrasting – 'before' and 'after' states of practice and points of view. In order for a study of student learning to use that state information, it must be represented in a form that is correlated with the salient features of transformative learning, and that allows comparisons among accounts. In other words, a representation scheme for that information must meet two conditions: well-defined correspondence to meaning frame operation, and standardized form.

Gurwitsch's framework satisfies the two conditions. His *universal formal pattern* for the organization of fields of consciousness comprises a standardized form, with well-defined correspondence to meaning frames (3.1.3). A characterization of student accounts as the contents of the three dimensions in a field of consciousness can act as a stand-in for the operation of a meaning frame, and thereby enable study of student transformative learning. Student experience of radically changed views and practice would be characterized as radically different 'before' and 'after' field of consciousness contents – most notably the presence in the 'after' field of elements unknown in the 'before' field – corresponding to the reformulated meaning frame of transformative learning.

**Aside:** Some computing education research evaluating how students understand software or systems depends on eliciting their mental and conceptual models (Ben-Ari *et al.* 2004). Meaning frame operation delineates what the models can – and cannot – encompass. Access to that operation (via fields of consciousness as stand-in representation) may support more informed elicitation.

#### 6.2.2 Introduction to Data Analysis

Data analysis proceeded in three steps:

1. characterization, in Gurwitsch's framework, of students' descriptions of (their relationship to) software and software development (a phenomenological analysis of student accounts, essentially);
2. comparison of 'before' and 'after' states characterized in step 1, to identify and evaluate occurrences of transformative learning (or not);

3. preliminary classification of results from step 2, based on patterns of similarity and difference, to determine and categorize types of transformative learning that occurred.

Combined results of steps 1 and 2 are found in 6.4 and 6.5. Step 3 was conceived as an initial, broadly defined classification of step 2 results, from which would proceed a comprehensive phenomenographic analysis (Booth 1997 p.138) of student experience according to the extent and nature of their transformative learning. Due to space constraints, this coarse partitioning was interrupted. A glimpse of in-process results is found in 6.6.

### 6.3 Transformative Learning: evidentiary data

Data in this section indicates the occurrence – in various ways – of transformative learning among CSX students.

#### 6.3.1 Operational Sequence

This student's account, excerpted from an end-of-term interview, closely matches Dewey's (4.2.1) operational sequence for *abstract thinking* / transformative learning.

(S\_m107): (prior expectations: ... *In this project, I'd thought the main focus was code.*)

i) a felt difficulty: *When we sat in the lab coding, and it wasn't working,*

ii) its location and definition: *I thought: there must be something to that module design document (I just happened to look at it while sitting in the lab). It said 'this invokes that' and we weren't doing it that way, and we were more focused on getting the code done.*

iii) suggestion of possible solution: *And I thought why did [the instructor] give us [these three weeks of other assignments] before code, if it's all about coding? I don't think you'd [the instructor] have given us all that time for other assignments [if it was all about coding]. ... Maybe it's not all about the code. ...*

iv) development by reasoning of the bearings of the suggestion: *It started giving meaning to me two weekends ago. The way I've always thought to do coding: ... I'd try this way; if that didn't work, I'd try that way. And if I created a wonderful piece of software in a course, and today I wanted to write it in another language, I couldn't do it. ...*

v) further observation and experiment leading to its acceptance or rejection; that is, the conclusion of belief or disbelief: *With the [design in documentation already done], you just have to worry about the final step of coding it in [any] language.*

#### 6.3.2 Eureka

These two student accounts, excerpted from end-of-term journals, are written from a perspective of reformulated meaning frames, the aftermath of transformative learning.

(S\_m201): *During this week, I have written the code based on the pseudo code and prior documents. It was amazing how quickly I was able to generate the code [1/2 hour] and how well it worked on the first run... Many*

*modules, including the infrastructure module (which initially seemed to be the most complex) ran flawlessly on the first run. ... I have really come to relate to [Parnas's] concept of faking the design process. ... [In the past, when] I'd work with a team, we'd develop documentation, and find later on [w]hat we forgot to anticipate, [and] I'd throw up my hands and jump into the code. [Later, the student explained that s/he would now return to the documentation as a design medium, to think through the new, unanticipated, issues, before beginning to code.]*

(S\_m202): *I was afraid the pseudocode would take me all week to do, so I started it on Monday of last week. It surprised me that I actually did the entire thing during Monday Night Football in about two hours, then revised it early on this week. [This student had remained skeptical of Parnas's approach throughout the semester. Her / his large programming project in another course was plagued with problems despite a month's hard work. Following that Monday night football game – one week before finals – s/he began the other project again, using an approach based on Parnas's work.]*

#### 6.3.3 Inside the Experience

Accounts from end-of-term interviews, one succinct, one expansive, document almost textbook examples of student experience as confusion → struggle → knowing.

Q: If you could change anything about this course, what would you change?

(S\_v101): *See, I don't know if I'd change anything; because I know, looking back on it, it sucked when you had to go through it. But looking back on it now, I can see why we did all the stuff that we did and the reasoning behind it.*

Q: If I'd asked you this question early in the semester, what would you have said [to change]?

*Everything*

Q: Looking back over the course, does it appear different to you at the end of the semester than at the beginning or middle? If so, how?

(S\_v102): *Very different ... [at the beginning of the semester,] it seemed like a piece of cake, no problem: read a book, write a program, and you're done, pretty much like any other programming class. ... At the middle [just before beginning the group project], I was kind of torn between two worlds - I still wanted to jump right into coding, but I had to force myself not to. ... You [the instructor] were pretty adamant about staying away from the computer and the compiler; Parnas was adamant too. ... I had to think about what we did earlier in the semester and really implement what we learned. ... I don't know how to say it; very much halfway between where I was at the beginning and where I was at the end. One side of me was saying 'Code', one side of me was saying 'Don't code'. I could see the point, but I didn't understand the picture. I understood each little point, but I didn't see how they fit together, until we started the project - or even the end of the project. At the end, that's when everything made sense, the big picture [came] at the end. It's important for future students [to know] ... 'Do not get discouraged; keep with it, it will make sense'. I wish I could just stand in front of people thinking about*

taking this class and say, 'Stick with it, it's tough but there's light at the end of the tunnel'. At the end now, looking back, I can say, 'Ah, now I see what [the instructor] was trying to teach us', but throughout the semester, that's not easy to see.

#### 6.4 Analysis 'Before' and 'During': only coding

As noted in section 5, students are strictly enjoined from coding until week 13. End-of-term interviews illustrate that virtually all students found it extremely challenging – if not impossible – to begin the project without coding.

Q: What did you find most difficult in the project?

(S\_m107): *Trying to change my way of thinking about approaching a software development project; it was difficult not sitting at a computer, I want to sit at a computer right away and code. You're doing something different than you've done in all your other courses.*

(S\_m104): *The coding [laughter] ...*

Q: Because you went to the coding right away?

Yes ...

Q: Did your ideas about the contents of the four [group project] assignments change over the course of the project? If so, how?

(S\_v102): *Yes, most definitely. ... We probably should have spent a lot of time on the documents, and then needed only a little time on coding. Instead, we spent a little time on documents, and a lot of time on coding. ... We all had a great, big headache on Monday night; we were so stressed out, we barely looked at the earlier documents ... Basically, what we've done in other classes: we've sat there, looked at the code, and given ourselves and each other headaches.*

Any one student's consistent behavior of this kind indicates his / her inability to hear instructions specifying non-coding approaches. Such 'deafness' evidences the power of an established meaning frame to effect selective awareness. Virtually every student in CSX behaves this way, signifying a virtually universal 'before' meaning frame: *It's all about code*. Analyzed from the perspective of Gurwitsch's work, feedback data demonstrates a pre-existing field of consciousness comprising the following content among its three domains:

- code occupies the thematic focus
- the thematic field is sparsely – if at all – populated
- all else within awareness is relegated to the margin

#### 6.5 Analysis 'After': motivation, alternatives

Two factors appear to play a major role in transforming students' practice from that described above: an enveloping context of quality (and the problematic consequences of poor quality software) as an ethical issue; and the capability to decompose the software development process in a way that manages complexity. The former enables them to see the problematic nature of an 'only coding' approach, and motivates them to consider other approaches. The latter makes alternative approaches possible. For many students, the two factors are deeply intertwined, with the acknowledgement (often explicitly stated) that what they knew – 'only coding' – does not work. Therefore, one set of feedback data, containing references to both factors, is presented here.

Q: Looking back over the course, does it appear different to you at the end of the semester than at the beginning or middle? If so, how?

(S\_v103): *Yeah, it's different from beginning to end; I guess the difference really comes in a fuller comprehension of the material or the subject matter. ... To be honest, I wasn't impressed [at the beginning], I was skeptical of ... [Parnas's] messages. At the midpoint, I started to realize there was truly some reality ... in the papers and in developing good quality software. ... [For example,] I gave some reasons on homework about why we could use [SDI] without a full nuclear war. But the more I think about it [Parnas's work], the more it just made sense, the more I believed it, the more I saw it in my mind's eye. ... Initially, [I believed that] anything can be done. If we get the job, it can be done. But thinking about Parnas's papers, why it can't be done ... [With a problem at work, I see such a difference between a] band-aid and fixing the real problem. ... [Before,] I thought [the best approach to take] was sheer persistence ... [After this semester,] I don't think it's the best way ...*

(S\_v102): *Much different. It seems like software design is actually a task that can be accomplished if you do it the right way. ... and it's definitely more about the process than the coding.*

Q: What will you take away ... from the course?

(S\_v101) ... *one thing I did learn in the course that I never even thought of before was that there were ethics behind computer science, and I didn't realize how many people out there abuse it. ... It just makes it so easy to deceive people sometimes, especially with the attitude people have now that technology can do everything. ... It can do a lot of cool stuff, but it can't do everything. ...*

Q: You've talked about the big role of documents, [However,] you spent [only] 4-5 hours on them, but 20-25 hours (total as a group) on code.

*If I'd not done the documents, started right away on the code without documents, it would have taken me a lot more time. That's because even though we weren't actually sitting down and writing the code, it was just like, as we did the documents, the picture of that code that I had in my head got more and more defined as we went down the chain. And then when we actually wrote the code, it's like my brain just dumped it all down on paper. I didn't even have to think about it; ... I didn't ... hit backspace, ... knew exactly [how] I wanted it to look.*

Q: That's unusual for you?

*Oh, yeah. Normally I'm ... going back, rewriting what I did before; and [the difference] was all thanks to the modularization. ... The reason it helped is that I knew where everything was and how it fit together. That's the hardest part of software development.*

To summarize the shift in content of students' three dimensions of the field of consciousness: Before, no possibility of managing complexity is found in any dimension; for some students, quality is not found in any dimension, and for the others, quality is relegated to the margin. Furthermore, the students had no sense of a development process; coding occupied the thematic focus as a prime element, to be addressed only by relentless effort, not decomposition and forethought.

After, both quality and conceptual approaches for managing complexity loom large in the (now well-populated) thematic field, each intermittently occupying the thematic focus. For students who fully completed group project assignments 10, 11, and 12 (before coding in assignment 13), code was relegated to the margin; for those who did not fully complete them, code resides as one element among others in this rich thematic field.

## 6.6 Glimpses: a long shadow

Implications of two almost identical puzzling statements preview a proposed comprehensive phenomenographic analysis of student transformative learning.

(S\_v103): *It just ... drove home to me how difficult it is to really develop robust software. [I didn't know it before].*

(S\_v102): *The reason [this course] is so tough is because it is **not easy** to develop good quality software. That's [a] huge [realization].*

How could they not have known the serious difficulty posed by software development? Neither could explain further. Review of the data with that question in mind strongly suggests that complexity was not to be found in any student's 'before' field of consciousness; it resided outside awareness. For these two, and others speaking more obliquely, transformative learning led to their becoming explicitly aware of it; it became the thematic focus of 'after' consciousness. I interpret that awareness as epistemologically prior to integrating into their practice Parnas's approaches to managing complexity. (Without awareness of a problem, its solution has no meaning.)

Based on the virtually universal relief, enthusiasm, and transformed practice among students when they begin to grasp Parnas's approach, I speculate that most of them experience a similar learning. Complexity appears to cast a long shadow, just below the level of their awareness, from early in students' practice of developing software.

## 7 Discussion

Three points informing the paper are viewed through a dual lens of transformative learning and software quality: rupture / crisis, uncertainty, and complexity.

### 7.1 Benefits of Rupture and Crisis

Rupture (for an individual) and crisis (for the community) play critical roles in intellectual development. Segal's explication of Heidegger's dynamic of rupture captures the nature and structure of discontinuity inherent in an individual's transformative learning. He describes *[e]xplicitness through rupture [as] the logic of the development of intuitions into publicly communicable forms ... Making explicit presupposes the ability to bring the shock of the not yet said but strongly felt into an explicit form.* (Note the 'long shadow' in section 6.6.) *To learn how to make our own and our students' practices explicit is therefore an essential part of the educational process* (Segal 1999 p.88).

Kuhn writes about *crises* as *necessary pre-conditions for the emergence of novel theories. ... [By] proliferating versions of the paradigm, crisis loosens the rules of normal puzzle-solving in ways that ultimately permit a new paradigm to emerge* (Kuhn 1996 pp.77,80).

## 7.2 Implications for Pedagogy

### 7.2.1 Experiential Learning

The power of a meaning frame to render aspects of the world unseen and to construe experience has significant implications for pedagogy. The classic sources express unanimity of opinion on how transformative learning happens: the meaning frame is reformulated only as a response to encounters with the world that result in persistent, significant observations that violate the frame; experiential learning is required.

### 7.2.2 Cultivate Well-Founded Uncertainty

Reflectiveness *depends most upon the capacity to suspend judgment and tolerate uncertainty: one must carefully determine the exact nature of the problem before proceeding to devise a solution* (Dewey 1991 pp.73-74). In computing education, courses are often taught as if every problem had a well-defined, known-in-advance solution. Since these conditions do not characterize software development, students cannot learn how to really develop software in such courses. As part of our responsibility as educators, we should allow students to enter – even choreograph their entry into – situations of uncertainty, and support them to find their way through it, while cultivating skills for reflectiveness.

Dewey advocated this position 100 years ago: *The difficulties that present themselves within the development of an experience are, however, to be cherished by the educator, not minimized, for they are the natural stimuli to reflective inquiry. Freedom does not consist in keeping up uninterrupted and unimpeded external activity, but is something achieved through conquering, by personal reflection, a way out of the difficulties that prevent ... spontaneous success* (Dewey 1991 pp.64,65). Booth, a contemporary scholar of computing education, agrees: *[P]roduction of working programs is no sign of an adequate understanding. Rather than assignments that can be solved by template programs, teachers should pose problems that allow interpretation* (Booth 1997 p.155).

### 7.2.3 Engagement and Stimulus, not Formula

Authentic reflective practice involves becoming aware of one's habitual behavior. It must come from a student's internal process, wherein questions arise out of dynamic engagement with the content. If self-observation is done by rote, it leads to confusion rather than insight: *[D]ogmatic commitment to observation produces a disengaged and decontextualised relationship to one's practice* (Segal 1999 p.75).

Similarly, the teacher's role cannot be specified in advance as a formulaic series of steps. Booth notes that imposing a pre-existing set of questions leads to

disastrous results (Booth, 1997 p.145). Dewey comments on information or observations communicated by the teacher to the student: It should include only content that the student could not readily acquire by personal observation; it should be offered in the form of a suggestion, a *stimulus, not with dogmatic finality and rigidity*; it should be made available only when it has relevance to the student's process. If done formulaically, it is not brought into a reflective process: [*Lying useless in the mind like*] debris, it is ... *an obstruction to effective thinking* ... (Dewey 1991 pp.198,199).

## 7.3 Implications for Curriculum Design

### 7.3.1 Received Meaning

Curriculum may be said to transmit a field of consciousness: Until one has developed a meaning frame for a particular content domain, one's received 'organization of consciousness' (the designation of significant elements and irrelevant elements with regard to that content domain) dictates how one navigates it. For the novice in an academic field, the curriculum dictates the element(s) to which one should direct attention (the thematic focus), and the elements that are noticed but deemed irrelevant to the point(s) of attention (the margin). The thematic field is defined as noticed aspects of the world relevant to the elements(s) being attended to. I speculate that when elements(s) being directly attended to are presented within a situating context, that context forms the content of the novice's thematic field. Without context, the novice's thematic field remains empty – and learning remains surface. Not least, curriculum dictates what is not to be noticed, what no one talks about.

This received field of consciousness in turn sets up the novice's meaning frame and approach to the topic from then forward (unless and until the frame is reformulated). Therefore, one can learn a great deal about a curriculum by examining students' meaning frames in the collective. Data in section 6 indicate that CSX students' pre-existing meaning frames had no place for either quality concerns or taking on complexity. One wonders if the curriculum to which these students had been exposed had no place for them either.

### 7.3.2 Challenge and Responsibility

While complexity – and the possibility of procedures to manage it – appear not to be found in any dimension of students' 'before' consciousness, they loom large in their 'after' understanding. I speculate that, lacking a widespread, established paradigm to manage complexity effectively, most educators don't address it directly. In developing software, the central challenge involves managing complexity (Peter (Naur 2007) described the core of computer science as *the scholarship of coherent description*); and the central responsibility involves formulating – and meeting – quality standards. I propose enlarging the discourse among educators and practitioners regarding these two critical areas, and building on that discussion to make them designated curriculum topics.

### 7.3.3 Parnas as Resource: confront complexity

CSX course content is based on work by David Parnas. Results from the course make clear that his work offers a strong foundation for addressing both the challenge of complexity and the responsibility of ethical concerns. Regarding the former, students develop new conceptual categories directly related to managing complexity; for example, (how) to use documentation as a real design medium. Regarding the latter, Parnas's explanation for resigning from the SDI Advisory Board (whether or not one agrees with his conclusion) offers a model of careful, professionally informed reasoning motivated by a sense of ethical responsibility and based on technical analysis.

## 8 Summary, Conclusions, and Future Work

### 8.1 Summary and Conclusions

This paper is written as part of an ongoing exploration into the operation and mechanisms of student reflective response (productively navigating the confusion created by an encounter with the unknown) – or not – and what is required to support it. Findings refine and accrue to previous results, expand the foundation on which future work will rest, and inform that work. I've investigated transformative learning so as to more clearly articulate the operation and mechanism of reflectiveness (itself the mechanism of transformative learning).

- Meaning frames *determine the essential conditions for construing meaning for an experience* (Mezirow 1991 p.44), and must satisfy (at least) two conditions to remain useful: 1) accuracy, correctly representing all aspects of the world considered relevant; and 2) availability below the level of awareness (thus inaccessible at a conscious level), internalized sufficiently to become operative (made manifest) without deliberate initiation. Student feedback data in 6.4 illustrates the power of an established meaning frame to effect selective awareness.

- In case of a rupture in knowing, transformative learning effects coherence in transition across that discontinuity; a reformulated meaning frame preserves meaning.

- In cumulative learning, old meaning is imposed upon new experience: pre-existing expectations are applied to interpret new experience. From transformative learning, new meaning arises, to be imposed upon new and old experience (Mezirow 1991 p.11). Experience anomalous to extant expectations (confusion) leads to reformulation of expectations (struggle); in turn, these direct interpretation of new and old experience (knowing).

- Kuhn's work in the history of science illuminates an analogous process (at community level) in developing esoteric bodies of discipline- or profession-specific knowledge. Terms for individual learning / *scientific group progress* correspond as follows: meaning frame / *paradigm*; cumulative learning / *normal science*; troublesome knowledge / *crisis*; transformative learning / *revolution*; reflectiveness / *extraordinary science*.

- Gurwitsch's work on the organization of consciousness offers indirect access to the operation of meaning frames, and clarifies distinctions among surface, cumulative, and

transformative learning. His description of awareness illuminates the role of context (as correspondent to a richly populated thematic field) in deep understanding.

In conclusion, the papers written thus far have introduced, investigated, and shown relevance of the literature to student experience of reflectiveness and defensiveness. Analysis of feedback data reveals students' 'before' states as lacking any real concept of software design process: no process, no design, no software, just bricolage and code; all the result of *surface learning* (Booth 1997 p.145). Consequences to software quality argue for designing curricula (perhaps, as in CSX, based on David Parnas's approach) that explicitly take on complexity, and for developing pedagogies aimed at both transformative and deep learning. The learner must be supported to discover for herself the *elusive obvious* (Feldenkrais 1981).

## 8.2 Future Work

Several possibilities exist for investigation: where in a course students collectively experience the dynamic of rupture; the learning of those few students who do not experience the dynamic (Heidegger's 'obliviousness'); the source of heightened affect during reflectiveness or transformative learning; a full phenomenographic analysis of CSX student feedback data for categories of transformative learning and 'before' / 'after' states.

Further, Kuhn's history of science strongly suggests that the discipline of computing remains in a pre-paradigmatic state. Investigation is merited into that possibility and (if true) what is required to mature the discipline beyond it.

## 9 Acknowledgements

I thank Raymond Lister for his thoughtful comments and intellectual generosity, Simon for formatting assistance, and the anonymous reviewers for their suggestions.

## 10 References

- Barer-Stein, T. (1987): Learning as a Process of Experiencing the Unfamiliar, *Studies in the Education of Adults*, **19**:87-108.
- Bauman, Z. (1990): *Thinking Sociologically*, Oxford: Basil Blackwell.
- Ben-Ari, M., Berglund, A., Booth, S., Holmboe, C. (2004): What Do We Mean by Theoretically Sound Research in Computer Science Education?, *ACM SIGCSE Bulletin* **36**(4), ITiCSE Conference Proceedings, panel session, Leeds, UK.
- Berkson, W., Wettersten, J. (1984): *Learning from Error: Karl Popper's Psychology of Learning*, LaSalle, IL, Open Court.
- Booth, S. (1997): On Phenomenography, Learning, and Teaching, *Higher Education Research and Development*, **16**(2):135-158.
- Boud, D. (2001): Using Journal Writing to Enhance Reflective Practice, *New Directions in Adult Continuing Education*, **90**(summer):9-18.
- Dewey, J. (1991 edition): *How We Think: a restatement of the relation of reflective thinking to the educative process*, Amherst, NY, Prometheus Books.
- Dreyfus, H. (1993): *Being-in-the-world: A commentary on Heidegger's being and time, division I*. MIT Press
- Feldenkrais, M. (1981): *The Elusive Obvious*, Capitola, CA, Meta-Publications.
- Gurwitsch, A. (1964): *The Field of Consciousness*, Pittsburgh, PA: Duquesne University Press.
- Heidegger, M. (1985): *Being and Time*, Oxford: Basil Blackwell.
- James, W. (1890): *The Principles of Psychology*, volume 1, Henry Holt, New York, NY.
- Johnson, D. G. (2001): *Computer Ethics* (3<sup>rd</sup> edition), Prentice Hall, Upper Saddle River, NJ.
- Kuhn, T. S. (1996): *The Structure of Scientific Revolutions* (3<sup>rd</sup> edition), University of Chicago Press.
- Loder, J. (1981): *The Transforming Moment: Understanding Convictional Experiences*, San Francisco, CA, Harper & Row.
- Mezirow, J. (1991): *Transformative Dimensions of Adult Learning*, San Francisco, CA, Jossey-Bass.
- Naur, P. (2007): Computing versus human thinking, Turing Award Lecture, *Communications of the ACM* **50**(1):85-94.
- Parnas, D. (2001): *Software Fundamentals*, Hoffman, D., Weiss, D., (editors) Upper Saddle River, NJ, Addison-Wesley .
- Sartre, J-P. (1966): *Being and Nothingness* (translation by Hazel E. Barnes), NY, Washington Square Press.
- Schwartzman, L. (2006): A Qualitative Analysis of Reflective and Defensive Student Responses in a Software Engineering and Design Course, *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling*, pp.46-53, Finland.
- Schwartzman, L. (2007): Student Defensiveness as a Threshold to Reflective Learning in Software Design, *Informatics in Education* **6**(1):197-214.
- Segal, S. (1999): The Existential Conditions of Explicitness: an Heideggerian perspective, *Studies in Continuing Education* **21**(1):73-89.
- Simon, H. (1996): *The Sciences of the Artificial* (3<sup>rd</sup> edition), MIT Press.

# ICT teaching and learning in a new educational paradigm: Lecturers' perceptions versus students' experiences

**Judy Sheard**

Faculty of Information Technology  
Monash University  
Caulfield East, Australia, 3145

judy.sheard@infotech.monash.edu.au

**Angela Carbone**

Faculty of Information Technology  
Monash University  
Caulfield East, Australia, 3145

angela.carbone@infotech.monash.edu.au

## Abstract

Over the last two decades there have been fundamental shifts in the way teaching and learning are perceived and conducted within the tertiary education sector. One is a move from teacher-centred to student-centred education, and another is a move from the traditional to the virtual classroom. However, there are indications that academics and students have not universally embraced this new educational paradigm. A further challenge faced by ICT educators is the rapidly evolving ICT discipline. In an ICT faculty at an Australian university, large-scale curriculum changes to the undergraduate degrees resulted in a set of foundational courses offered to all students in their first year of study. With such major changes, it seemed timely to investigate students' and lecturers' perspectives of the new degrees. Using interviews, similarities and differences between students' and lecturers' perceptions of their educational program were found. The findings are presented under the themes of motivation, interest, course content, learning environment and adaptation. A conclusion of this study is that students and lecturers operating within a student-centred, technology-enhanced educational paradigm are facing difficulties, and there is a need for universities to work towards addressing these issues.

**Keywords:** First year experience, lecturer and student perceptions, ICT teaching and learning, student-centred technology-enhanced, educational paradigm.

## 1 Introduction

The tertiary teaching and learning environment is a complex system within which students and educators interact to foster learning. Important for understanding this system are the perspectives of both students and educators. The students' interests and experiences of their educational environment influence their approach to learning and ultimately their learning outcomes (Berglund & Eckerdal, 2006; Berglund & Wiggberg, 2006). A number of studies have shown that educators' perceptions

of their students as learners influence their understanding of teaching roles and responsibilities. This may be reflected in the educational environment they provide. An educator's perception of student motivation, interests and capabilities may, to varying degrees, influence their interactions with students, their teaching approach and the way they present the curriculum (Kember, 1997; Prosser & Trigwell, 1999; Samuelowicz & Bain, 1992).

Investigating students' and educators' perspectives may be useful for informing curriculum design or pedagogical approaches and may also help address misconceptions about course content or the teaching and learning environment (Greening, 1998). Any mismatch between students and their educators in understandings of teaching and learning can result in frustrating experiences (Fox, 1983). Reconciling these misconceptions is critical for the provision of an educational environment that accommodates the needs and interests of the students. This can result in a more satisfying educational experience for students and assisting their adaptation to the tertiary environment (Milliken & Barnes, 2002). As Trigwell and Prosser (1997) propose:

...teachers need to be aware that they do not experience the same world, and students do not experience that same world as that which teachers have designed ... a major task of teaching is to work towards developing teaching and learning situations which students experience in similar ways to that which the teacher has designed. (p. 250)

Although there have been many studies of student and educator perspectives in tertiary education, very few studies were found that investigated both. In one of the few complementary studies found, Hoppes and Chesbro (2003) explored the commonalities and differences in views on instruction between educators and students. They found disagreement in the values and beliefs held about teaching techniques and approaches. These were explained in terms of teacher-centred perspectives of the educators versus learner-centred perspectives of the students. Hoppes and Chesbro propose that studies such as these can allow educators "to reflect critically on presuppositions held about elements of teaching effectiveness" (p.167). Furthermore, they allow an opportunity for educators and students to understand each other's views and work towards common goals.

The purpose of the study presented in this paper was to investigate lecturers' perceptions of their students'

motivation, interests and adaptation in the first semester of an ICT degree and the alignment of these perceptions with the students' experiences. This study was part of a research project that aimed to establish student and staff perceptions of ICT degrees and consequential influences on student and lecturer behaviour in the teaching context. The main objectives of the project were to inform and enhance current teaching practice within the Faculty's undergraduate teaching programs and to improve information provision to potential students and their advisers.

The project was largely motivated by a major restructuring of an ICT faculty's undergraduate offerings. This resulted in a set of common foundational courses delivered across multiple degrees and on local and international campuses. Each course was developed by a team of academics. All course materials are made available on a learning management system. The courses are delivered mainly in face-to-face mode; however, each course has a small cohort of off-campus students. Most students study these degrees in fulltime mode. Typically it is their first attempt at tertiary study, with approximately two-thirds entering directly from secondary school.

This paper is organised into six sections. Section 2 presents an overview of previous studies that have explored ICT tertiary teaching and learning. Section 3 describes the context of the study, the study design and data collection methods. Section 4 reports the results, highlighting the areas of alignment and mismatch between students' experiences and lecturers' perceptions. This is followed in Section 5 by a discussion of the difficulties that students and lecturers face in the current tertiary ICT education system. The paper concludes with directions for further research.

## 2 The ICT tertiary teaching and learning landscape

Teaching in the ICT discipline at the tertiary level presents many challenges. A tension ICT academics commonly face is whether to teach specific skills, which may help students gain immediate employment, or teach foundational material to better prepare students for long-term employment in the ICT industry (Dall'Alba, 2005). The rapidly changing ICT landscape means that academics are regularly under pressure to update curriculum and teaching materials. With frequent updates to degree programs it is difficult for potential students and careers advisors to keep abreast of these changes. As Soh, Samal and Nugent (2007) remark:

"Ever-changing technology also makes it difficult to provide high school students with a consistent and up-to-date coursework foundation. As a result, college level introductory CS courses are filled with students with diverse knowledge and exposure" (p. 60)

Thus, it is not surprising that a study by Multimedia Victoria (2007) found that 48% of school students believed that they did not know much about ICT study at tertiary institutions. In addition, almost a quarter stated

that it was difficult to find out about ICT degrees at tertiary institutions. This suggests that some students enrol in ICT degrees without a clear idea of what is involved in studying ICT, and this may result in a less than satisfactory educational experience.

These discipline-specific issues are in addition to challenges facing academics in general. Within the tertiary education sector over the past two decades there has been a fundamental shift in the way teaching and learning are perceived and conducted. Arguably, there are two major factors in this change — the first is the move from teacher-centred to student-centred education, and the second is the move from the traditional to the virtual classroom, largely enabled by the Web and its related technologies (Lefoe, 1998; Verbeeten, 2001-2002). Harasim (2000) proposes that these changes have resulted in a *new educational paradigm* that features new modes of delivery, electronically enhanced or constructed learning environments, and changed roles and responsibilities for students and educators. As Barone (2005) remarks, "Technology and pedagogy are converging in the learning landscape. Often this collides with the process, structure, governance, power relationships and culture values of the traditional campus" (p14.1).

However, there are indications that many academics have not embraced the new educational paradigm. Greenberg (2004) claims that "Teaching and learning tend to be served up in the same old containers, in the same old spaces, using the same old concept of face-to-face interpersonal relationships" (p.13). Barone (2005) maintains that, although academics may be unhappy with the formal classroom setting, most have not made the transition to alternatives offered by the new educational paradigm. For example, although academics may use the Web to provide resources to their students, they still maintain the traditional face-to-face teaching as they see this as ensuring "that students are learning the 'right' things the 'right' way" (p14.6). The higher education sector is still bound by traditional values and expectations. As Entwistle, Skinner, Entwistle and Orr (2000) argue, the pedagogies that academics adopt are strongly influenced by their own experiences as students. Another issue is that very few academics have the opportunity to engage in discourse in which they could reflect upon their teaching experiences.

From another perspective, there is evidence that many students are not comfortable with the new educational paradigm (Roberts, 2005; Windham, 2005). Although most students entering university today have not known a world without computer technology, there are indications that many students prefer only a moderate level of technology use in their educational environment. A study by Kvavik (2005) found that while students expressed a desire for some level of online delivery, almost all (97.8%) still wanted face-to-face interaction. Another study by Roberts (2005) found that, when asked to rate their preference for the level of technology use in a lecturing environment, all 25 college students voted for 50% lecturing and 50% interactive technology. These studies and others indicate that there may not be complete

truth in the supposition that educators need to use technology to appeal to their students. Another issue is that although the current students have far greater skill levels than previous generations of students, they may not have the skills required in the areas necessary to support their academic work (Roberts, 2005).

With these changes in the ICT tertiary sector, and large scale curriculum changes within our Faculty, it seemed timely to investigate students' experiences of their ICT degree and learning environment, and the lecturers' perceptions of their students, and consider these in terms of teaching practice.

### 3 Research approach

This study was conducted at five of the Victorian campuses of Monash University in four undergraduate ICT degrees of the Faculty of Information Technology: *Bachelor of Information Technology and Systems*, *Bachelor of Business Information Systems*, *Bachelor of Computer Science* and *Bachelor of Software Engineering*. With recent changes to these degrees, students in first year now study a set of foundational courses that are common to all degrees. This section presents details of the research design, data collections and participants.

#### 3.1 Study design

The study was conducted by a team of six researchers from the Computing Education Research Group at Monash University. The data collections for the study were conducted in first semester 2006 as follows:

- mid-semester interviews of lecturers who were teaching or co-ordinating a first year ICT degree;
- mid-semester interviews of students in the first year of an ICT degree.

The lecturer interviews were designed to explore lecturers' perceptions of their students and the influence of these on their teaching practice. The student interviews were designed to explore the students' expectations and experiences of their degrees and learning environment.

Further data collections of a start-of-semester survey and final results provided an indication of the factors influencing students' performance and progression. The findings from this data are reported elsewhere (Sheard, Carbone, Markham, Hurst, Casey, & Avram, 2008).

The study was approved by the Monash University Standing Committee on Ethics in Research involving Humans (SCERH).

#### 3.2 Lecturer interviews

Lecturers were interviewed following a semi-structured interview schedule. The interviews were conducted during the middle of first semester 2006. All lecturers involved in teaching the first-year foundational courses were invited to participate. Seventeen lecturers were interviewed from approximately 30 lecturers who were invited.

A set of questions for the semi-structured interview were prepared by the project team. The questions were designed to gather information about lecturers' understanding of their students' background and knowledge, and how they approach their teaching practice and curriculum development.

The duration of the interviews ranged from 30 to 70 minutes, approximately. The interviews were audiotaped.

#### 3.3 Student interviews

The students were interviewed following a semi-structured interview schedule. The interviews were conducted during the middle of first semester 2006.

Students were recruited for the interviews when the survey was administered. A total of 119 students agreed to be interviewed and a selection of these students were later contacted by email to arrange an interview time. The selection aimed for a spread across the five Victorian campuses, with 4-6 students per campus.

Twenty-five students were eventually interviewed from 31 students who were selected. There were 22 male and three female students aged from 18-25. The proportion of male students was higher than in the first-year student cohort. Most of the participants were local students (21 out of 25) with English being first language (20 out of 25). Their ENTER<sup>1</sup> scores varied from 39.15 to 96.70.

A set of interview questions was prepared by the project team. The focus of these questions was to determine the students' reactions to their degree and to the teaching and learning context. The interview questions were mostly open-ended to encourage the students to raise issues of interest or concern.

Twenty-one interviews were conducted face-to-face at the student's home campus, and four interviews were conducted by phone. The interviews were up to 20 minutes in duration, and were audiotaped.

#### 3.4 Analysis

The analysis of the interview data was a three-stage process. First, the analysis of the lecturer interviews was conducted using an interpretive approach. The interview data was searched for comments relating to the lecturers' perceptions of their students' interests and expectations. Five themes emerged: *motivation*, *interest*, *course content*, *learning environment* and *adaptation*. These themes provided a framework for analysis of the student interviews. Second, the student interview data was searched for comments relating to each theme. This enabled explorations of areas where the students' experiences of their ICT degree were in agreement or conflict with lecturers' perceptions. Third, the lecturer interview data was further searched for influences on their teaching programs.

---

<sup>1</sup> Equivalent National Tertiary Entrance Rank. This is the tertiary entrance score in the Australian state of Victoria.

## 4 Results

The findings report the students' and lecturers' comments from the interview data using the framework that emerged from analysis of the lecturer interviews: *motivation, interest, course content, learning environment and adaptation.*

### 4.1 Motivation – Why are they here?

Lecturers claimed that students' motivation for doing their ICT degree was an important factor in learning outcomes. Many lecturers maintained that the most common source of motivation for students was the desire for a career in ICT. A couple of illustrative comments:

*The majority are looking for a job. Although the university is focused on research and research students, the majority of our students will not head in that direction. ... They hope for career and job opportunities.*

*Students want a job at the end of a [multimedia degree] and they are very focused. It is a difficult issue because an undergraduate degree is a generalist degree and not a specialist degree; the idea is that the faculty is building fundamental skills in ICT but not turning out individuals who are multimedia specialists. Students find that difficult because they are looking for the 'best of the best' in multimedia and [want to] come out and step straight into a job with high level skills. So their focus is very much career-focused.*

However, the lecturers' perception of students as being career-motivated was not supported by the student interviews, where *only one* student commented on a future career in ICT:

*I enjoy the course because I have a reason to come to the course. I know what I want to pursue in the future.*

A couple of lecturers remarked that some students did not know why they were doing their degree:

*Many students are fairly naïve, and they have not thought about why they want to do ICT or considered their motivations for doing it. Some students do an ICT degree because it is the only degree they can get into.*

*Some students do put in a lot of research about their course options before they enrol, but it depends. This varies a lot. A lot of students do think quite carefully about the sort of career they are interested in. On the other hand some are less career-focused and do not give it [much] thought.*

A number of lecturers raised concerns about students' lack of motivation. Some lecturers perceived this as more of a problem than lack of ability to master the course content. As a couple of lecturers explained:

*Students are definitely able, in terms of my field [databases]. New students are motivated at the tutorials, the lectures, they are working away. No*

*doubt they can do it. Motivation is the only concern, and some of them do not have it.*

*A significant problem is student application, and if they are not motivated they will not apply themselves to their study. ... There are many examples reflecting the sorts of problems students have applying themselves to the course, and it is not a case of them having the academic mind to study.*

The lecturers' view of the students as career-motivated clearly influenced teaching practice. A number of lecturers mentioned that they tried to motivate students by linking the course material to their future work. For example:

*When lecturers are teaching foundational material they try to explain the direction in which it will lead and why the students are learning this material ...*

However, a couple of lecturers expressed concerns about the effectiveness of this strategy for first year students:

*... this [linking course material to future careers] is often lost on the students.*

*It is the transition from first year to third year where students' application and discipline to their study and how subjects [courses] are delivered can change. Students are conscious of the fact that they will soon have to go and get jobs so they are more interested in being proactive.*

### 4.2 Interest – What do they want to learn?

Lecturers claimed that another important factor in students' success was their level of interest in their degree and the course content. Interest was often associated with motivation, as one respondent commented:

*[Students] come in and think they can get a good job in this area but they're not particularly interested in the subject matter so they're less likely to succeed.*

Lecturers perceived that the students' level of interest varied across the courses, and this was a particular problem with mandatory foundational courses:

*Often what is of interest to students is not the core [foundational] material which they need to understand. Students have a hard time coping with the backbone of the subject, i.e. the basic building blocks. Students are all keen to learn the applications and the process in which these things are used, but the drudgery of having to go through and learn the fundamentals is something they are not altogether interested in and do not cope well with. That makes it difficult and a challenge for teaching.*

*Common core [foundational] courses are going to create problems because different student cohorts are not going to have the same interest in the subject. ... If they are forced into that situation they will not respond as well as they could academically if they apply themselves based on personal choice and interest.*

An issue reported by many lecturers concerned catering for the needs of students with different levels of interest and ability in the mandatory courses taught across multiple campuses and into different degree programs:

*Lecturers have to start with the basics. However they cannot spend too much time on the basics and must assume that students that have not done any work in these areas will spend additional time catching up on that material. This is because most of the students who have started that course have an interest in that area and have therefore exposed themselves to it through their own personal actions or through subjects that they have done in high school.*

*Lecturers cannot expect tutors to hold the rest of the class back for these students, and may have to get the tutor to give them special attention or separate lessons.*

#### 4.3 Course content – What do they expect to learn?

Some lecturers claimed that students had little knowledge of their degree when they enrolled:

*There are more students who are unaware of what is in their course than are aware of what is in it. This is particularly true of course selection, in that they do not know what they are going to learn.*

*Students enrol in ICT because they hope it is going to be of interest to them, and because they have no experience that is usually all they can do — hope.*

This perception was supported by the student interviews where most students could not articulate a clear understanding of course content. This could lead to a favorable outcome, if the course content was more interesting than expected, or disappointment, if expectations were not met. Some illustrative comments from students:

*I didn't clearly know my expectation before doing the degree because I only decided to do it one month before coming to uni, but I was pretty excited about it. I guess I expected it to be more about using ICT in business, more about business aspects than technical aspects.*

*The course has not yet met my expectation. I actually expected it to be more involved in networking as the name suggested. It came to the point that I felt some of the classes were pointless. I am not learning anything new with the way the course was structured.*

*So far I have been pleasantly impressed about the stuff that we are doing in the [courses]. I have actually learned more than what I have expected to learn. The degree is challenging enough. The content has been brilliant. All of the lecturers and tutors are very approachable.*

Meeting students' expectations and interests was an obvious area of concern for lecturers. A number of

lecturers stressed the importance of students researching the degree offerings before enrolment so they know what to expect; however, some lecturers perceived that many students were not proactive in this respect:

*Students see a course name and synopsis but they have little idea of what is actually involved in it. They do not do any pre-reading and they do not understand the jargon used in the handbooks, for example.*

For one student not knowing what to expect from a course resulted in discontinuing the degree:

*I think computer systems needs to be promoted more because when I was in year 12, not many people know what it was. If I were told more about what I would be doing in this course, I wouldn't be so panic. ... I would still be doing that course... [this student had discontinued from the course just prior to the interview]*

#### 4.4 Learning environment – Where do they want to learn?

Both lecturers and students maintained that tutorials provided more valuable learning situations than lectures. Many lecturers commented on the low level of attendance at lectures. For example:

*Attendance rates at lectures are abysmal. Ideally it would be better to move away from lectures and towards a method with one staff member per group of twenty students, and then teach to these small groups for the entire semester. There is a cost issue, of course.*

*Few students download and read the web published lecture notes before lectures. Many fail to show up to lectures and/or tutorials with pens, show an unwillingness to put pen to paper and would rather communicate via email.*

In the interviews the students gave insights into their attendance at classes. Some typical comments:

*Attended the first few lectures, I found them difficult to follow. I found that it was more effective for me just to skip the lectures and do the readings with the lecture's guide. Sitting in the classes to listen to the lectures without an engaging environment did not encourage learning. During the tutorials is where I learned the most. Tutorials provide the educational environment that is similar to high school where students are expected to actively contribute to classes and tutors actively help students.*

*... I think tutorial sessions are useful. I think they are more helpful than the lectures. I actually learn more in tutorial sessions.*

*Some lectures are hard to follow. I can't gain understanding on some materials. I like to learn by example, better than straight theory. I absorb it really well that way.*

Only a couple of students gave indications that they attended all classes:

*...I go to tutes and classes. I don't miss any classes and that helps me a lot. When I go to lectures, I understand better, and then I have more time left to do other things. I make sure that I don't miss anything. But many students, they don't really come to class.*

*All the lecturers are good. I am fortunate that I haven't got any lecturer that reads from lecture's note.*

Most lecturers were unhappy with the students' level of attendance, claiming that this made a difference to their ultimate success. Lecturers had various strategies to encourage their students to attend classes. These included personalising the class environment and making it necessary to attend class in order to collect course materials. Many lecturers believed that posting materials online was a definite factor in low attendances.

*At the start of semester, walk around the lecture theatre for about 5-10 minutes as students are arriving and talk to them very informally, especially in the first few lectures as this is where staff and students can get to know one another. Students would especially benefit from this approach because they may feel less overwhelmed and intimidated by lecturers as a result. When lecturers do this the gap between the students and staff should be very much reduced and students will 'open up'.*

*Avoid posting tutorial solutions on the web ....*

However, a number of lecturers commented that students appeared to be satisfied with the level of interaction with lecturers and their learning environment. This was supported by the student interview data with many commenting on the approachability and accessibility of the lecturers. As one student remarked:

*I like the teachers ... because they are helpful. You can raise your hand to stop them and ask questions in the classes ...*

#### 4.5 Adaptation – How have they adapted?

Lecturers claimed that students were academically able, and would succeed if they made a genuine effort, but that many students were not prepared for the independent style of learning expected at university. As these lecturers explained:

*Most students have not got the faintest idea what they are in for ... secondary schools just do not have the knowledge and experience to prepare the students for what university life is going to be like.*

*Although students are academically able they have to learn a completely different mode of thought (independent thinking), and they are not prepared for that at all.*

*At school they were being driven by parents and this is the regime they were used to. Making that adjustment from school to university where they are responsible for their learning can be difficult. It is their process at this stage, not the lecturer's.*

The comments made by students during the interviews supported these perceptions. For example:

*Totally different environment than high school. It's a lot more like you are on your own a little bit. Unlike in high school that you are having someone and the same classes every day. It's interesting. It made you feel independent. It's good.*

*Changing from high school, I found myself settled quite well. The hours are a lot less restricted. Workload is lighter, have more time left. I have more choices. High school at this time had a lot of workload. I just started university, so the workload is not much now. Don't see much different from high school. I am just required to be more independent.*

*More difficult than high school, more workload, more difficult things to learn. A lot less guidance compared to high school. They only give you the weekly topic and you need to prepare before coming to class. The guide is very broad, they don't tell you specific things to study for exam. Need to do a lot of preparation.*

A number of lecturers commented that students from other cultures experienced particular problems:

*Many Asian students have a very strong concept of 'this is the answer', and to be able to say that there are many, different ways of doing something and that they are all quite reasonable, and getting the students to express their opinions is a real challenge for many lecturers. These students often want the solution rather than going through with the discussion beforehand. Once they are comfortable this does change.*

*In terms of ICT students' academic ability, some international Asian students struggle with poor English language skills and comprehension.*

A couple of international students remarked on such problems. For example, one commented:

*I need to work harder. There is more stress in uni than in high school. For me the ICT things are new. I have to spend time to go through all the slides, [course] guides and the readings.*

Although lecturers recognised that students needed assistance in meeting expectations of the university study environment, a number expressed dissatisfaction as to the level of help that students appeared to need:

*Students can do very little and yet many courses have guides written to help them which are 'almost kindergarten standard', but students seem almost to need that very basic repetition. It helps them. Over the years some lecturers' expectations have reduced dramatically. Students seem to need or else expect the material at 'real building block style level'. They seem to be able to 'jump off' by themselves after that*

From the students' perspective, they appreciated clearly stated expectations. As one student commented:

*It's really help when the subject tells you ahead of time what you need to do, what to read and what kind of assignment, how many hours you should spend on reading and so on. I liked that in ICT course, whereas in commerce they don't do that.*

## 5 Discussion

Many studies have indicated that educators need to develop a better understanding of their students as learners; as Barone (2005) proposes, “fundamental to the ability to transform the academy is the wisdom and humility to know students, their motivations, their goals, and their learning styles” (p14.13). Our study is a small step in that direction, exploring educators’ perceptions of their students within a student-centred, technology-enhanced educational paradigm. This research provides insights into students’ experiences and behaviour, in the environment that we, as educators, have provided, and highlights the difficulties faced in *transforming the academy*.

In our study, many lecturers expressed an awareness of student-centred learning and its potential benefits to students. As one lecturer member described succinctly:

*If students know what learning is all about and what it means to them they have few problems. Learning means being reflective, learning is life-long and continues after the two hours of lectures and two hours of tutes a week. Attitude is very important, as is students’ approach to learning. Transfer of facts, memorisation, exam sitting, none of this is learning. Students need to learn to grasp concepts and internalise their learning; they can then re-use it any time, in any situation. Learning is about long-term retention of knowledge*

Despite showing a good understanding of student-centredness, educators were experiencing difficulties operationalising this ideal. While educators continued to teach in physical spaces designed for teacher-centred instruction, they were also required to provide and manage online learning environments. With the ready availability of online course materials, many students did not see the need to attend class. While educators saw their online learning environment as *supplemental* to classes, students, by contrast, used it as a *replacement* for classes. Access to course materials online allowed students to work largely independently, studying and learning away from the physical teaching space, and without the educators’ guidance. This was a great concern for educators, as they perceived that fewer students were engaging with learning. As a consequence, many lecturers made considerable efforts to keep their students coming to class, but this was often ineffective.

A theme of the student-centred educational paradigm is the changed roles of students and educators. An ideal view of student-centredness is the students taking responsibility for their own learning with guidance from their educators. Our study found that first-year students varied in their readiness and willingness to accept an independent learning approach. Overseas students

especially were seen as having difficulties adopting a learning approach suitable for tertiary study.

Lecturers perceived that a career in ICT was a strong motivation for many students. Consequently, they saw a difficulty in teaching foundational courses in a generalist ICT degree where there were no obvious links to a career. However, in the interviews, only one student commented that career was a source of motivation, while most claimed that their motivation came from interest in the content of their particular degree. This indicated a misalignment in student and lecturer perspectives. The lecturers were focused on addressing extrinsic sources of motivation, while students were intrinsically motivated. This suggests that for first-year students, a career is not foremost in their thinking; and efforts by lecturers to link course material to a future career may not be as useful in motivating students as trying to address their interests.

Lecturers showed a willingness to adapt curriculum and teaching approaches to meet the needs and interests of their students; however, the nature of the degree program provided little flexibility for this to occur. With the foundational courses taught over multiple campuses, and in on-campus and distance education mode, there was a requirement that materials were prepared well before the semester commenced, leaving little opportunity for changes to suit different student cohorts. Another difficulty was that course materials were prepared in the context of competing demands of different individuals and degree structures. Input from multiple lecturers had in some cases resulted in bloated or severely diluted courses that were standardised across degrees with different foci and learning outcomes. Curriculum development under these conditions was necessarily a compromise, making it impossible to cater for the variety in students’ interests and abilities. As one educator expressed:

*...And at the moment the faculty are mandating that students will all do the same subjects. The effect of that is that the subject needs to be reduced down to such a level that it has to satisfy both, and in the process important content is excluded.*

Lecturers found courseware development under these conditions demanding and stressful, and perceived this as satisfying efficiency and quality measures rather than pedagogical needs. These competing demands left educators with a strong sense of disempowerment.

## 6 Conclusions and Future Work

There have been fundamental shifts in the tertiary education landscape, with the move from teacher-centred to student-centred learning and from the traditional to the online learning environment. In ICT, these changes have been compounded by a fast-moving discipline that necessitates frequent course revisions, leaving students uncertain about their course content and educators uncertain about how to accommodate their students’ learning needs. In our study, institutional rules and program course structures, involving multi-campus, on and off campus delivery, meant that educators had limited

flexibility to respond to the needs and interests of their students.

The student-centred, technology-enhanced paradigm has influenced teaching practice and the ways in which students approach their learning. However, in our study there were indications that educators and students were having difficulty achieving the ideals of this educational paradigm. We found tensions between the expectations and behaviours of educators and students. The *educators*, operating within the time and space constraints of the physical campus, appeared stuck in a teacher-centred delivery mode while also having to provide an online learning environment. The *students*, faced with the freedom of anytime, anyplace, anyhow learning enabled by a technology-enhanced learning environment, were often choosing to work independently and away from the guidance and influence of their educators. Although online technology may be seen to facilitate student-centredness by allowing students to take more responsibility for their learning, the way students were choosing to use it appears in conflict with this core tenet of student-centred learning. The findings from our study indicate that this is an area requiring future research.

## 7 Acknowledgements

We acknowledge the contributions by other members of the project team: Selby Markham, John Hurst, Chris Avram and Des Casey, and the research analysts Varintra Tui and Victoria Heathcote. We thank the students and lecturers for participating in the data collection. We also acknowledge the financial support from the Associate Dean (Education) in the Faculty of Information Technology, Monash University, Australia.

## 8 References

- Barone, C. (2005). The New Academy. In D. Oblinger & J. Oblinger (Eds.), *Educating the Net Generation*. Washington DC.: e-EDUCAUSE.
- Berglund, A., & Eckerdal, A. (2006). What do CS students try to learn? Insights from a distributed, project based course in computer systems. *Computer Science Education*, 16(3), 185-195.
- Berglund, A., & Wiggberg, M. (2006). *Students learn CS in different ways. Insights from an empirical study*. Paper presented at the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2006), Bologna, Italy.
- Dall'Alba, G. (2005). The role of teaching in higher education: Enabling students to enter a field of study and practice. *Learning and Instruction*, 3, 299-313.
- Entwistle, N., Skinner, D., Entwistle, D., & Orr, S. (2000). Conceptions and beliefs about "Good Teaching": an integration of contrasting research areas. *Higher Education Research & Development*, 19(1), 5-26.
- Fox, D. (1983). Personal theories of teaching. *Studies in Higher Education*, 8(2), 151-163.
- Greenberg, M. (2004). A University Is Not a Business (and Other Fantasies). *EDUCAUSE Review*, 39(2), 10-16.
- Greening, T. (1998). *Computer science: Through the eyes of potential students*. Paper presented at the Australasian Computer Science Education Conference (ACE 1998), Brisbane, Australia.
- Harasim, L. (2000). Shift happens: Online education as a new paradigm in learning. *The Internet and Higher Education*, 3, 41-61.
- Hoppes, S., & Chesbro, S. (2003). Elements of instruction in allied health: Do faculty and students value the same thing? *Journal of Allied Health*, 32(3), 167-172.
- Kember, D. (1997). A reconceptualisation of the research into university academics' conceptions of teaching. *Learning and Instruction*, 7(3), 255-275.
- Kvavik, R. (2005). Convenience, Communications and Control: How students use technology. In D. Oblinger & J. Oblinger (Eds.), *Educating the Net Generation*. Washington DC.: e-EDUCAUSE.
- Lefoe, G. (1998). *Creating constructivist learning environments on the Web: The challenge in higher education*. Paper presented at the Fifteenth Annual Conference of the Australian Society for Computers in Learning in Tertiary Education (ASCILITE 1998), Wollongong, NSW, Australia.
- Milliken, J., & Barnes, P. (2002). Teaching and technology in higher education: student perceptions and personal reflections. *Computers and Education*, 39, 223-235.
- Multimedia Victoria. (2007). ICT Skills Research: Attitudes to ICT careers and study among 14-19 year old Victorians (years 9-12) <http://www.mm.vic.gov.au/AttitudestoICTcareers> Accessed 23 July 2007.
- Prosser, M., & Trigwell, K. (1999). Relational perspectives on higher education teaching and learning in the sciences. *Studies in Science Education*, 33, 31-60.
- Roberts, G. (2005). Technology and learning expectations of the net generation. In D. Oblinger & J. Oblinger (Eds.), *Educating the net generation*. Washington DC.: e-EDUCAUSE.
- Samuelowicz, K., & Bain, J. D. (1992). Conceptions of teaching held by academic teachers. *Higher Education*, 24, 93-111.
- Sheard, J., Carbone, A., Markham, S., Hurst, J., Casey, D., & Avram, C. (2008). *Performance and progression of first year ICT students*. Paper presented at the Tenth Australasian Computing Education Conference (ACE2008), Wollongong, Australia.
- Soh, I.-K., Samal, A., & Nugent, G. (2007). An integrated framework for improved computer science education: Strategies, implementations and

results. *Computer Science Education*, 17(1), 59-83.

Trigwell, K., & Prosser, M. (1997). Towards an understanding of individual acts of teaching and learning. *Higher Education Research and Development*, 16(2), 241-252.

Verbeeten, M. J. (2001-2002). Learner-centred? It's just a click away... *Journal of Educational Technology Systems*, 30(2), 159-170.

Windham, C. (2005). The students' perspective. In D. Oblinger & J. Oblinger (Eds.), *Educating the net generation*. Washington DC.: e-EDUCAUSE.



# Koli Calling Comes of Age: an Analysis

Simon

School of Design, Communication, and Information Technology  
The University of Newcastle  
Newcastle, Australia

simon@newcastle.edu.au

## Abstract

A detailed analysis of the full papers presented at the first six years of Koli Calling shows how the conference has matured over that time to become more research-oriented. While the first three years were dominated by papers proposing or reporting on classroom initiatives, the next three have seen a remarkable increase in the proportion of papers describing experimental and analytical research. This paper quantifies and explores that increase, and considers in addition the range of topics, contexts, and scopes of these six years of papers.

**Keywords:** Computing education research, literature review, classification.

## 1 Introduction

Koli Calling was launched in 2001 “to develop the exchange of relevant information between colleagues working within the same discipline” (Sutinen & Kuittinen 2002). In the manner typical of conferences, there was a list of seven indicative topics, and that list included Computer Science Education Research.

By the fourth year there was much talk of “CS Education and CS Education Research”, and it was decided to categorise each paper as either *discussion* (“papers that present novel ideas, approaches, and systems for CS Education”) or *research* (“papers in which these issues have been elaborated further in some rigid research setting”) (Malmi 2004).

The fifth year elaborated on these definitions and added system and poster categories:

- research: submissions presenting a novel approach, method, tool, finding, interpretation, explanation, or other contribution in a solid scientific framework;
- system: papers describing methods or tools for learning or instruction in CS or a related field, presented in a constructive framework with references to related work;

- discussion papers and posters: typically describing original work in progress (Salakoski 2005).

Salakoski also noted that “over the years, [Koli Calling] has developed into its present form of a rigorous scientific meeting . . . Last year can be considered as the breakthrough of the conference” (Salakoski 2005).

At the 2006 conference the category definitions took on a slightly different hue:

- research: presenting novel results, methods, tools or interpretations that contribute to solid, theoretically anchored research;
- system: tools for learning or instruction in computing education, motivated by the didactic needs of teaching computing;
- discussion: presentation of novel ideas and prototypes;
- poster: work in progress (Berglund 2007).

In addition, in 2006 the conference name changed to reflect the increased research emphasis. Formerly known as the Finnish / Baltic Sea Conference on Computer Science Education, it now became the Baltic Sea Conference on Computing Education Research..

While Salakoski was quite certain that 2004 marked a watershed in the research emphasis of Koli Calling, the variation in the number and definition of categories of papers leads to three clear questions:

1. What is a research paper?
2. What proportion of papers presented at Koli Calling can be called research papers?
3. Has the proportion of research papers increased over the six years of Koli Calling?

I have recently devised a classification for computing education papers (Simon 2007), and now report on the application of this classification to the past six years of Koli Calling, to address these three questions and possibly to discover other features of interest.

The next section summarises some prior classification systems that have been applied to computer education research or to the publications that report on such research. The paper then proceeds to explain my own system, report the results of applying it to Koli Calling, compare those results with other categorisations of Koli Calling papers, and draw some conclusions.

## 2 Prior Classifications

The literature includes a number of systems for classifying papers in computing education. In Simon (2007), an analysis of recent papers at two Australasian conferences, these systems are discussed at some length, and attention is devoted to explaining why they were not suitable for the purpose of that analysis. Here they will be presented rather more briefly.

Perhaps the best known and most widely cited is Valentine (2004), who surveyed 20 years of SIGCSE Technical Symposium papers dealing with first-year Computer Science subjects, putting each paper into one of six categories:

- experimental: papers including any sort of scientific analysis;
- Marco Polo: descriptions of the application of a new curriculum, language, or course;
- philosophy: attempts to generate debate on philosophical grounds;
- tools: software tools developed to assist with aspects of teaching/learning or assessment;
- nifty: innovative, interesting ways to teach abstract concepts;
- John Henry: papers describing outrageously difficult ways of undertaking simple tasks.

While this categorisation distinguishes between research papers (the experimental category) and others (the remaining five categories), it does have some weaknesses, not least of which is the lack of clear guidelines as to how subsequent researchers might apply the scheme.

Fincher and Petre (2004) proposed ten subfields for computing education research, and Pears, Seidman, Eney, Kinnunen, and Malmi (2005) combined these into the four broader fields of

- studies in teaching, learning, and assessment,
- institutions and educational settings,
- problems and solutions, and
- computing education research as a discipline.

However, neither of these classifications appears to distinguish between research papers and others.

Randolph, Bednarik, and Myller (2005) examined the full papers in the first four years of Koli Calling, performing a thorough analysis of the methodology of the 17 papers that involved research with human participants. The study pays scant attention to the remaining 42 papers, but does categorise them as

- literature reviews, meta-analyses,
- program descriptions without anecdotal evidence,
- program descriptions with anecdotal evidence,
- theoretical, methodological or philosophical papers,

- technical investigations, and
- other.

It might be tempting to conclude that the 17 papers represent research while the 42 do not, but this is clearly inappropriate, as research papers can certainly be found among, for example, meta-analyses and methodological papers.

Until now, therefore, it might appear that Valentine's 'taxonomy' is the only system that attempts to distinguish between research and non-research papers, and the basis of that distinction is not entirely clear.

Furthermore, all of these systems tend to linearise, to place upon a single axis properties that are in fact orthogonal. For example, Valentine's experimental category addresses the nature of the paper, while his tools category addresses the paper's subject matter. This is my main reason for choosing to devise a new classification, with four orthogonal dimensions, which I here apply to all 102 papers published in the proceedings of Koli Calling between 2001 and 2006.

## 3 A New Classification for Computing Education Papers

The new classification system categorises papers according to four distinct dimensions, which are orthogonal in the sense that each is independent of the others, and that a paper's categorisation in one dimension imposes no constraints or limitations on its categorisations in the others. The nature of a paper describes what sort of paper it is; its topic describes what it is about; its context describes the subject matter of the course in which it is based; and its scope is a measure of the breadth of the work in the computing education community.

The system was devised with the purpose of forming an overall picture of the papers presented in recent years at the two major Australasian computing education conferences. It categorised so as to summarise, so as in turn to facilitate an overview of the research. The development of the system is described in more detail in Simon (2007).

### 3.1 Nature

Many computing education researchers distinguish between practice and research papers. This new system of classification further divides research papers into two separate categories, *experiment* and *analysis*, introduces a *report* category that is reasonably congruent with practice papers as they are generally understood, and notes and explains an additional category, *position* papers.

According to prior definitions already canvassed, research can suggest "some rigid research setting" (Malmi 2004), "a solid scientific framework" (Salakoski 2005), "contribut[ing] to solid, theoretically anchored research" (Berglund 2007), or "including any sort of scientific analysis" (Valentine 2004). While these definitions suit the purposes for which they were intended, I have chosen to be rather more specific.

I define an *experiment* paper as one that reports on a clear and deliberate research experiment. The authors have set out to answer a particular question, devised a study (which might be as simple as a survey) to assist in that regard, carried out the study, gathered the data, and analysed it. An example paper in this category is *A multi-national, multi-institutional study of student-generated software designs* (Fincher, Petre, et al 2004).

An *analysis* paper is one whose authors have set out to answer a particular question, gathered existing data as appropriate, and analysed it. Unlike an experiment paper, it does not involve devising and conducting a study; instead it uses data already available, such as existing class results, the literature, or other sources. A typical analysis paper is *Progress Reports and Novices' Understanding of Program Code* (Mannila 2006).

A *report* is a paper describing something that has been tried or developed in an educational context. Many papers of this sort exhibit the minimal analysis of conducting a student survey to confirm the appearance of success; so long as it is clear that the principal intent of the paper is to report on the trial or the development, I still classify such papers as reports. A paper that falls clearly into this category is *Producing interactive web lectures with authorware* (Kerola 2003).

Finally, a *position* paper is one that elucidates the authors' thoughts on a matter, or perhaps sets out plans for future work, without having anything concrete to report upon. An example of a position paper is *Fibonacci Numbers Using Mutual Recursion* (Rubio & Pajak 2005).

By these definitions, I believe that both experiment and analysis papers would normally be regarded as research, that reports would normally be considered practice papers, and that position papers are in a field of their own.

It has been suggested that papers describing innovations in computing education constitute research, and that it is therefore unreasonable to exclude all reports from the body of work identified as research papers. This position can certainly be argued, but so can the position that any paper reporting on anything of interest to the computing education community constitutes research. The delineation in this system is inescapably subjective, but it does have the advantage of drawing a line between research and practice papers, and indeed position papers; and such a line is necessary if one is to have any chance of answering questions such as whether the proportion of research papers at Koli Calling has increased over the years.

Even so, more investigation is required into exactly what does constitute research, and into the validity of the research versus practice division. In the meantime, for the purposes of this paper it is probably safe to suggest that experiment and analysis papers are definitely research, and to leave open the question of whether some or all reports and position papers also qualify for that description.

I have been asked why the system distinguishes between experiment and analysis papers when both are combined

to form what I am calling research papers. The answer to this is that the system was developed not to identify research papers but to analyse a corpus of papers and see what emerged. When the analysis was being performed, these two categories seemed quite distinct (and easily distinguished), and were both therefore incorporated into the system. It happens that together they comprise a more widely recognised category, but this does not diminish the distinction between them.

### 3.2 Topic

The topic dimension describes what a paper is actually about, and its membership emerges from a content analysis of the corpus being studied. Even so, the list of categories appears to be reasonably stable: the list that emerged for Koli Calling, shown in Table 1, is all but identical to the list that emerged from the prior study of two Australasian conferences (Simon 2007).

Most of the topics should be reasonably self-explanatory, but a few of them might require a word of explanation.

With regard to both *assessment* and *teaching/learning*, the *tools* topic is used for a paper reporting on the development of a new tool, or perhaps a novel use of an existing tool, while papers that report on a reasonably expected use of an existing tool will come under the *techniques* category. *Do students SQLify? Improving learning outcomes with peer review and enhanced computer assisted assessment of querying skills* (de Raadt et al 2006) reports on the development of a new tool for use in assessment, and so is categorised under *assessment tools*; whereas *Automatic grading of graphical user interface programs exploiting Jemmy* (Surakka et al 2005) reports on the use of existing tools to perform assessment tasks, and so is categorised under *assessment techniques*.

While *teaching/learning techniques* concerns ways of teaching and learning (for example, *Learning programming by programming: a case study* (Hassinen & Mäyrä 2006)), *teaching/learning* concerns the act of teaching and/or the act of learning (for example, *Survival of students with different learning preferences* (Bednarik & Fränti 2004)).

The *research* topic does not indicate whether a paper is a research paper – the nature dimension does that. In the topic dimension, *research* denotes papers that are essentially *about* research – as, for example, Randolph et

**Table 1: topics covered in six years of Koli Calling**

ability/aptitude	ethics/professional issues
assessment techniques	gender issues
assessment tools	language/culture issues
cheating & plagiarism	recruitment
credit for prior learning	research
curriculum	teaching/learning
distance/online delivery	teaching/learning techniques
educational technology	teaching/learning tools
employment	tutors & demonstrators

al (2005).

### 3.3 Context

Some readers might be surprised not to see topics such as first-year programming, capstone projects, group work, and so on. After careful consideration I have decided that these are very seldom the topic of a paper; rather, they are the *context* in which the work was done and the paper written. This therefore forms a new dimension, which I call context. A paper's context will most often be a subject area of some sort, such as programming, computer systems, theory of computation, etc; but not all computing education papers are set in the context of particular subjects, so further categories, such as literature, have been added to better reflect the observations.

As with topic, the values in this list will vary according to the corpus of work being analysed. Table 2 lists the contexts found while analysing the six years of Koli Calling papers. Again, most of these values should be fairly self-explanatory. A broad-based context describes a paper that is set in no particular subject area, either because it covers multiple subjects (*Contextual computing studies in Tanzania* (Sutinen et al 2002)) or because it is very general and mentions no particular subjects (*Evaluation of faculty workload for various methods in computer science education* (Kurhila 2002)). Many capstone projects entail group work; a paper dealing with such a project will be categorised according to whether the emphasis is on the project subject itself (*Moral conflicts perceived by instructors of a project course* (Vartiainen 2005)) or on the groups undertaking the project (*A pilot study concerning power in CS student project groups* (Wiggberg 2006)).

### 3.4 Scope

The final dimension helps describe the breadth of the work on which the paper is based. The narrowest focus is a single *subject*; some papers report on a range of subjects within a *department/program*; others might focus on the whole *institution*, and others again on *many institutions* (where 'many' can be as few as two).

Not all papers have an identifiable scope. *Explanograms: low overhead multi-media learning resources* (Pears & Olsson 2004) explains a tool that can be used to help with teaching, but the explanation is not based on a particular

subject, on many institutions, or on something between. For this reason, the scope dimension includes a *not applicable* category.

The principal value of this dimension is that it can be seen as one possible measure of collaboration within the computing education community: while work in a single subject can be the work of an individual or a small teaching team, work across multiple institutions necessarily reflects significant community involvement. This is why the dimension is retained in the system despite the fact that it cannot be usefully applied to all papers.

## 4 Classifying Koli Calling

Having established the classification system described in section 3, it seemed reasonable to apply it to other computing education conferences, both to facilitate an overview of the papers at those conferences and (eventually) to permit some sort of comparison of the major computing education conferences worldwide.

This classification system has been applied by a single researcher to all 102 full papers published in the proceedings of Koli Calling 2001-2006, with consideration where appropriate to trends over that time. 'Full papers' is here taken to mean all papers published in the proceedings other than those labelled as keynote, invited, demo, or poster.

As mentioned before, the system was not devised with any particular agenda other than to categorise, summarise, and view; but in the case of Koli Calling, it did seem to offer a means of assessing the suggestion that the conference had become more research oriented, so the analysis was carried out with that suggestion in mind.

### 4.1 Nature

Taking the definition of nature in section 3.1 as an answer to what constitutes a research paper, what proportion of Koli Calling papers are research papers, that is, papers whose nature is either experiment or analysis?

Table 3 shows the number and proportion of papers that fall into each of the categories of nature. Combining the experiment and analysis categories, we see that about 35% of all full papers are classified as research papers. This is comparable with the recent analysis of the major computing education conferences in Australian and New Zealand (Simon 2007), in which 22% of the papers were categorised as experiment and 13% as analysis, for the same 35% overall proportion of research papers.

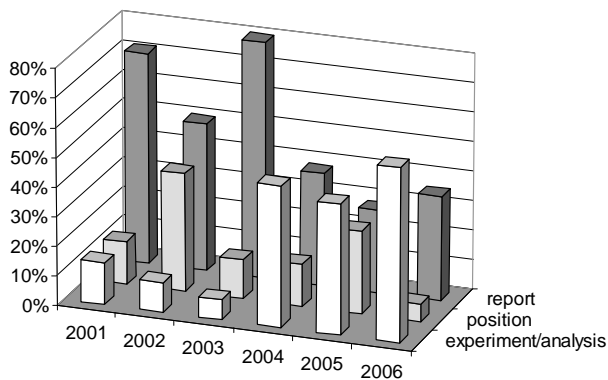
The third research question of this paper concerns a possible increase of the proportion of research papers

**Table 2: contexts found in six years of Koli Calling**

broad-based	literature
capstone project	logic
compilers	mathematics
data structures	programming
database	software engineering
group work	study planning
hardware/architecture	theory of computation
information systems	work experience
introduction to IT	writing

**Table 3: natures of all full papers**

	count	proportion
experiment	14	14%
analysis	21	20%
report	48	47%
position	19	19%



**Figure 1: Proportion of papers by nature and year**

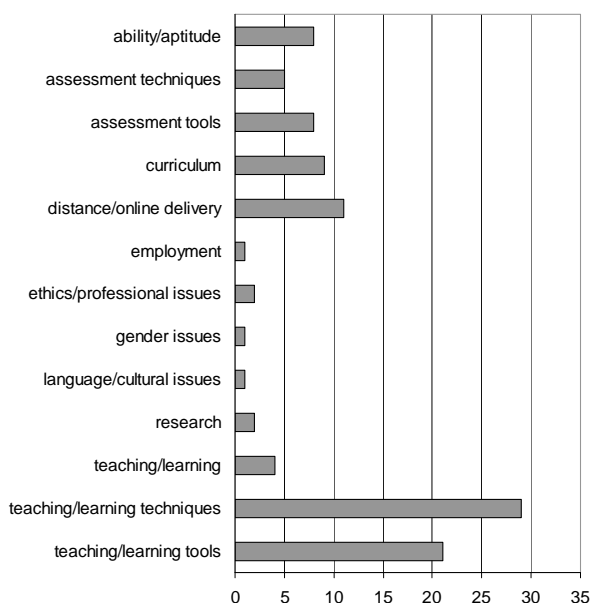
over time. Figure 1 represents the data graphically, combining experiment and analysis papers into a single research grouping, which is plotted alongside position and report papers for each year of the conference.

The results here are little short of astonishing. For the first three years of the conference, research papers made up respectively 14%, 10%, and 7% of the proceedings; the next year they surged to 47%, a level that was all but maintained with 44% in 2005 and surpassed with 59% in 2006.

Salakoski (2005) did not have this classification system to hand when he described 2004 as a year of breakthrough, but this analysis clearly supports his evaluation of the 2004 and 2005 conferences, and indicates that the pattern has continued at least to 2006.

## 4.2 Topic

Figure 2 shows the number of papers in each topic over the six years of Koli Calling. Given the ‘swap meet’ nature of computing education conferences, it is not surprising to see a large number of papers on teaching/learning techniques (28%), teaching/learning tools (20%), assessment techniques (8%), and assessment tools (8%). A solid 8% of papers on ability/aptitude is



**Figure 2: number of papers in each topic**

more or less guaranteed by the perennial problem of programming aptitude, while the strong representation of papers on distance/online delivery (11%) might reflect a particular facet of Finnish education, as it is almost twice the 6% found in the earlier study of Australian and New Zealand papers (Simon 2007).

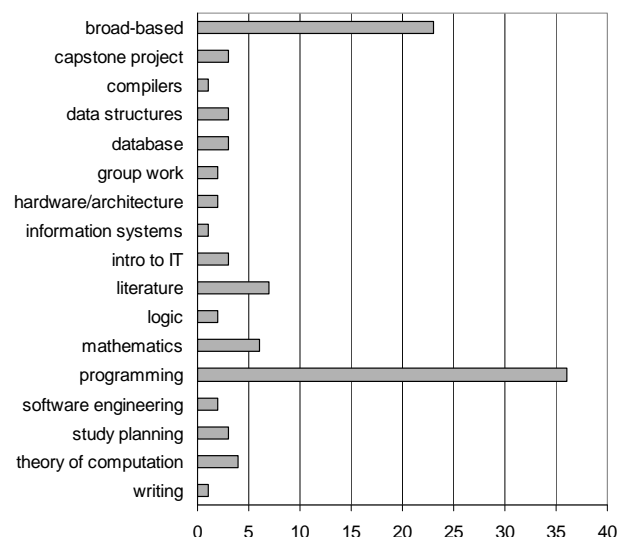
## 4.3 Context

Figure 3 illustrates the number of papers in each of the observed contexts. The predominance of work set in the context of programming subjects (35%) is to be expected, perhaps because so many computing education subjects entail programming, and almost certainly because of the constant attempts we make to address the particular difficulties faced by students in learning to program. Programming aside, the only standout context is broad-based, at 23%, and that is because this category does not represent an actual context but encompasses papers that cover more than one context and more abstract papers that are devoid of context.

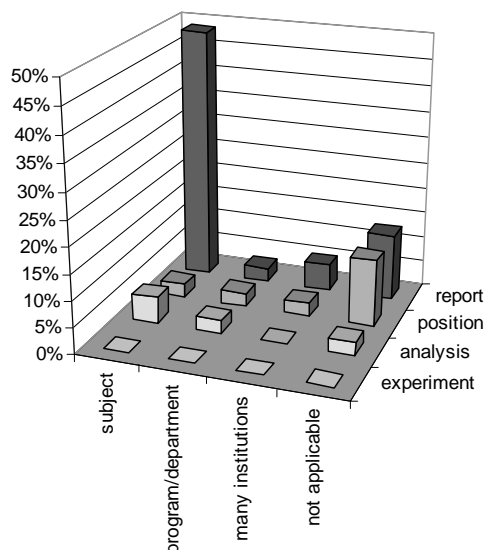
## 4.4 Scope

Over the six years of the conference, some 53% of the papers have reported on work concerning a single subject; 11% on work spread across a program or department; 9% on work concerning multiple institutions; and 27% on work with no identifiable scope.

As mentioned in section 3.4, I contend that work conducted across multiple institutions tends to indicate a stronger involvement with the computing education community than work dealing with single subjects. In the prior study using this classification system (Simon 2007) I observed some correlation between nature and scope: essentially, multi-institutional papers are more likely to be research papers than reports or position papers. Having noted the 2004 surge in research papers at Koli Calling, I was curious as to whether the scopes of those research papers bore out this apparent correlation. Did the increase in experiment and analysis papers correspond with a broadening of scope?



**Figure 3: number of papers in each context**



**Figure 4: scope and nature of papers, 2001-2003**

Figure 4 shows a two-dimensional analysis of papers by nature and scope for the first three years of the conference, and figure 5 shows the same analysis for the next three years. While both figures are clearly dominated by reports based on single subjects, this dominance dropped from nearly 50% in the first three years to about 25% in the next three.

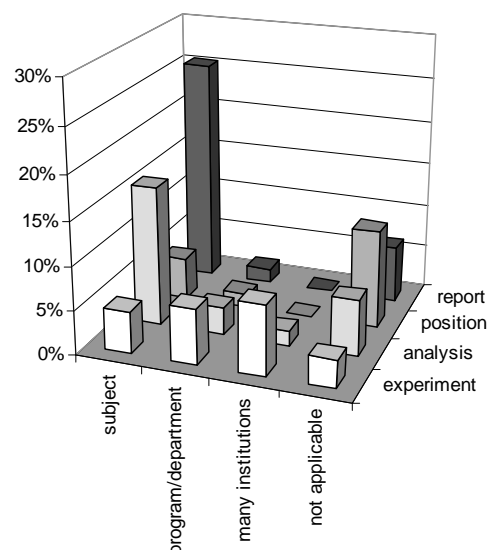
Having already observed the surge in experiment and analysis papers in the three years from 2004, we now see from figures 4 and 5 that many of the new research papers are also based in single subjects (5% of the papers in these three years are experiment papers in the subject scope and 16% are analysis papers in the subject scope), but the 9% that are research papers based across the program/depart (6% experiment and 3% analysis) and the 10% that are research papers based across multiple institutions (8% experiment and 2% analysis) show that the surge in research papers does indeed correspond with an increase in papers of broader scope.

## 5 Comparison with Prior Work

How do these findings tally with those of Randolph et al (2005) in their study of what was essentially a single major aspect of the papers from 2001 to 2004? Unfortunately, this is not as straightforward a question as it seems, because the differences between the systems are too great.

Randolph found that 17 papers involved research with human participants while 42 did not. Over the same period I find 14 experiment and analysis papers and 46 report and position papers. While I cannot explain the minor difference in the total, I believe that these findings are reasonably congruent.

As the main thrust of Randolph's categorisation concerns methodology, his system counts methodological cases rather than papers, and identifies 74 cases in the 59 papers. It seems, though, that most of the multi-case papers are in the human research area, as the remaining 42 papers appear to give rise to only 44 cases. Therefore it might seem reasonable to compare my counts of non-



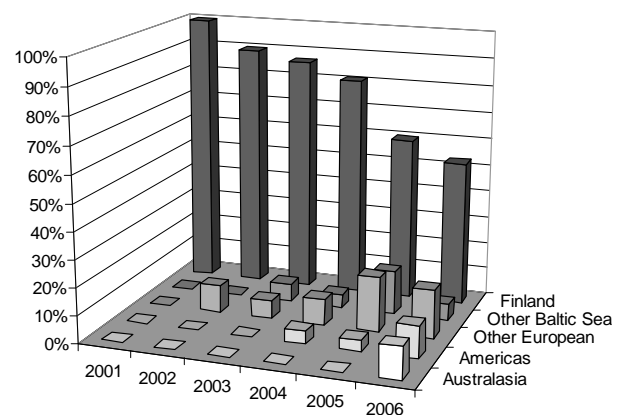
**Figure 5: scope and nature of papers, 2004-2006**

research papers with Randolph's counts of non-human-research cases – if I could work out what to compare with what.

Randolph's 'literature reviews, meta-analyses' might well fall into my analysis category; 'program descriptions without anecdotal evidence', 'program descriptions with anecdotal evidence', and 'technical investigations' might generally correspond to my reports; and 'theoretical, methodological or philosophical papers' might correspond to my position papers. But these putative correspondences are rather too vague to be of use, and after a brief look at the numbers I have concluded that no meaningful comparison is possible.

As an aside, Randolph noted the region of first author's affiliation for each paper, finding that about 90% of all papers came from Finland. Extending this analysis to the following two years and examining it year by year, I note a steady internationalisation of the conference (figure 6); the number of papers whose first authors are from Finland has dropped from 100% in 2001 to 53% in 2006.

It is interesting that there are consistently fewer papers from Baltic Sea countries other than Finland than there are from the rest of Europe. I leave the conference organisers to speculate on the reason for this apparent



**Figure 6: internationalisation of Koli Calling, as shown by affiliation of first author**

lack of interest from Finland's close neighbours.

In addition to Randolph's study, it might be worth looking for some congruence between this paper's nature classification and the research/system/discussion distinction introduced for the more recent Koli Calling conferences. One might expect that most of the papers classified in the proceedings as research papers would fall into my experiment and analysis categories, and that most of the papers classified in the proceedings as technical investigations would fall into my report category.

Figure 7 presents the cross-tabulation of the nature classification with what I shall call the proceedings category. We must bear in mind that these categories were introduced only in 2004 and were added to in 2005, so there are many Koli Calling papers that have no proceedings category at all.

Nevertheless we see that of the papers examined, most of those whose proceedings category is research are indeed experiment (28%) or analysis (45%) papers, and that most of those whose proceedings category is system are indeed reports (88%). It is also no surprise that position papers make up a solid 35% of the papers categorised in the proceedings as discussion papers.

What might be more puzzling is that the papers categorised in the proceedings as discussion papers include 23% that I categorise as experiment papers and 15% that I categorise as analysis papers. This is explained by the fact that discussion papers are solicited somewhat differently from research or system papers; they are clearly intended to be reports on work in progress, and to be shorter than research and system papers. Therefore this difference is not a shortcoming of Koli Calling's discussion paper category or of my classification system; rather, it is a recognition that work in progress can fall into any of the four nature categories.

## 6 Conclusions

This classification system for computing education papers incorporates a reasonably firm definition for the notion of a research paper, which can fall into one of two categories. An experiment paper is one that reports on a research question, a study to address that question, and an analysis of the results of the study, while an analysis paper is one that reports on a research question and an

analysis of existing data to address the question.

It is pleasing to report that some 35% of the full papers presented at Koli Calling fall into one or other of the research categories, and that the proportion displayed a remarkable increase in 2004 and has sustained that increase since then.

This increase in the proportion of research papers could be due either to an increased proportion of research submissions or to an increased inclination to accept research submissions over others. The two published Koli Calling acceptance rates (Malmi 2004, Salakoski 2005) are quite high, so it seems unlikely that the change is a result of the paper selection process; one must conclude that the conference really has seen a dramatic increase in the proportion of research papers being submitted.

### 6.1 Research Papers versus Practice Papers

How can the computing education community benefit from this work? While many academics appear to be aware of a distinction between practice and research papers, a clear explanation of the distinction might help them to target their work to the nature that they prefer. I believe that the concrete definitions presented here will be of help in this regard.

In addition, I believe that a clear understanding of this classification system will assist conference chairs to better distinguish the natures of submitted papers, which might help if the chairs wish to lead the conference in a particular direction, such as towards a higher proportion of research papers.

While this paper might lean slightly towards the position that research papers are in some sense better than practice papers, that is not its intention. The primary intention was to present and apply a tool that can be used to summarise, and therefore to better view, a large corpus of papers in computing education research; and a secondary intention was to see if the resulting view provided any support for earlier assertions that Koli Calling had developed into a more research-oriented conference.

### 6.2 Future Directions

To date there has been no study of inter-rater reliability in the use of this classification system. Such a study is expected to take place at a workshop planned for early 2008. In addition, it is planned to analyse other major computing education conferences over a comparable timeline, with a view to comparing and contrasting those conferences.

It is clear that there is also scope for further investigation into the apparent division of papers into research and practice; into the validity of labelling experiment and analysis papers as research and labelling reports and position papers as 'not research'; and into such value-laden areas as a desirable balance between research and practice papers.

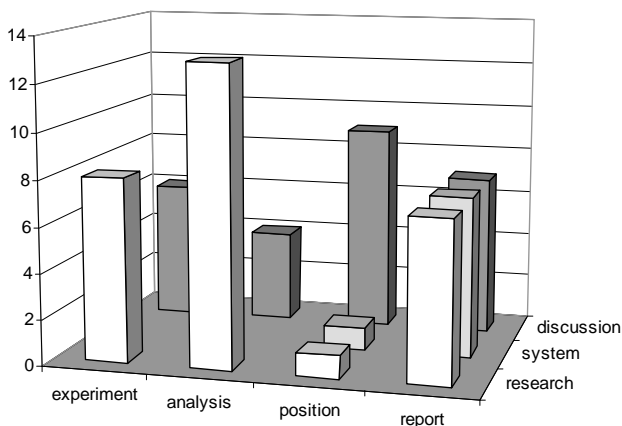
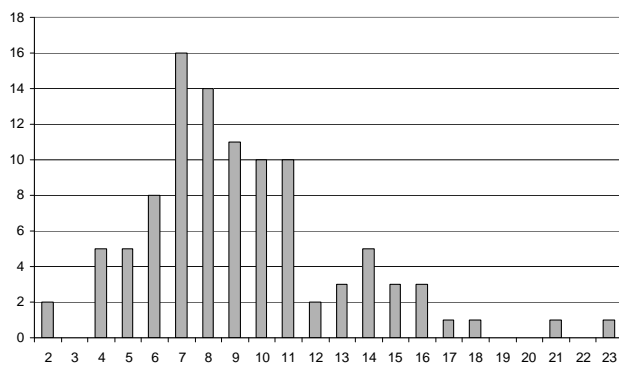


Figure 7: nature vs proceedings category



**Figure 8: number of words in the titles of papers**

### 6.3 Postscript

As a minor distraction from the analysis of the papers I also recorded the number of words in the title of each. The results are presented in figure 8. Are submissions with seven-word titles more likely than other submissions to be accepted to Koli Calling? Readers are invited to draw their own conclusions.

### Acknowledgements

This paper has improved as result of feedback from the Koli Calling referees, discussion with many people at the conference, and subsequent discussion with Leslie Schwartzman. I am grateful to all of these people.

### References

- Bednarik, R. & Fränti, P. (2004). Survival of students with different learning preferences. *Proc. 4th Finnish / Baltic Sea Conference on Computer Science Education*, 121-125.
- Berglund, A. (2007). Foreword to *Proc. 6th Baltic Sea Conference on Computing Education Research*, iii.
- de Raadt, M., Dekeyser, S., & Lee, T.Y. (2006). Do students SQLify? Improving learning outcomes with peer review and enhanced computer assisted assessment of querying skills. *Proc. 6th Baltic Sea Conference on Computing Education Research*, 101-108.
- Fincher, S., & Petre, M. (2004). *Computer science education research*. London, Routledge Falmer.
- Fincher, S., Petre, M., Tenenberg, J., Blaha, K., Bouvier, D., Chen, T-Y., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R., Monge, A., Moström, J.E., Powers, K., Ratcliffe, M., Robins, A., Sanders, D., Schwartzman, L., Simon, B., Stoker, C., Elliott Tew, A., & VanDeGrift, T. (2004). A multi-national, multi-institutional study of student-generated software designs. *Proc. 4th Finnish / Baltic Sea Conference on Computer Science Education*, 20-27.
- Hassinen, M. & Mäyrä, H. (2006). Learning programming by programming: a case study. *Proc. 6th Baltic Sea Conference on Computing Education Research*, 117-119.
- Kerola, T. (2003). Producing interactive web lectures with authorware. *Proc. 3rd Finnish / Baltic Sea Conference on Computer Science Education*, 18-21.
- Malmi, L. (2004). Foreword to *Proc. 4th Finnish / Baltic Sea Conference on Computer Science Education*, iii.
- Mannila, L. (2006). Progress Reports and Novices' Understanding of Program Code. *Proc. 6th Baltic Sea Conference on Computing Education Research*, 27-31.
- Pears, A., & Olsson, H. (2004). Explanograms: low overhead multi-media learning resources. *Proc. 4th Finnish / Baltic Sea Conference on Computer Science Education*, 67-74.
- Pears, A., Seidman, S., Eney, C., Kinnunen, P., & Malmi, L. (2005). Constructing a core literature for computing education research. *ACM SIGCSE Bulletin*, 37(4) 152-161.
- Randolph, J., Bednarik, R., & Myller, N. (2005). A methodological review of the articles published in the proceedings of Koli Calling 2001-2004. *Proc. 5th Finnish / Baltic Sea Conference on Computer Science Education*, 103-109.
- Rubio, M., & Pajak B. (2005). Fibonacci Numbers Using Mutual Recursion. *Proc. 5th Finnish / Baltic Sea Conference on Computer Science Education*, 174-177.
- Salakoski, T. (2005). Foreword to *Proc. 5th Finnish / Baltic Sea Conference on Computer Science Education*, iii.
- Simon (2007). A classification of recent Australasian computing education publications. *Computer Science Education* 17(3) 155-169.
- Surakka, S., Auvinen, J., & Ihantola, P. (2005). Automatic grading of graphical user interface programs exploiting Jemmy. *Proc. 5th Finnish / Baltic Sea Conference on Computer Science Education*, 49-56.
- Sutinen, E., & Kuittinen, M. (2002). Foreword to *Proc. 1st Annual Finnish / Baltic Sea Conference on Computer Science Education*, ii.
- Sutinen, E., Vesisenaho, M., & Virnes, M. (2002). Contextual computing studies in Tanzania. *Proc. 1st Annual Finnish / Baltic Sea Conference on Computer Science Education*, 84-88.
- Valentine, D. (2004). CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. *Proc. 35th SIGCSE Technical Symposium on Computer Science Education, ACM SIGCSE Bulletin*, 36(1) 255-259.
- Vartiainen, T. (2005). Moral conflicts perceived by instructors of a project course. *Proc. 5th Finnish / Baltic Sea Conference on Computer Science Education*, 25-32.
- Wiggberg, M. (2006). A pilot study concerning power in CS student project groups. *Proc. 6th Baltic Sea Conference on Computing Education Research*, 132-135.

# Students' Understandings of Storing Objects

Juha Sorva

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

## Abstract

This paper reports a phenomenographic study of how introductory students view objects that have been created and stored by an object-oriented program. By analyzing student interviews, we identify five categories of description, each representing a different kind of understanding of the phenomenon. Of these categories, some represent viable understandings that we would like our students to have. Others are partially incorrect and indicate that some students mistakenly focus their awareness on aspects that are unhelpful or even harmful for constructing a viable mental model of storing objects. This paper brings together two previously disjointed branches of computer science education research: the study of misconceptions and the phenomenographic research approach. The phenomenographic approach used in this study extends traditional phenomenography by including partially incorrect understandings in a phenomenographic outcome space, and explicitly treating them as such. This approach offers a new way of studying misconceptions and linking them to correct understandings of a phenomenon.

**Keywords:** objects, students' understandings, misconceptions, phenomenography, constructivism, CS1

## 1 Introduction

### 1.1 Background

Object-oriented programming is complex and abstract. According to constructivist theory, beginning programmers construct different kinds of mental models of the underlying layers of abstraction (Ben-Ari 2001). Some of these student-constructed models will be 'correct' or at least viable in practice, some of them will be unorthodox but relatively harmless in terms of skill development, and some of them will be just plain wrong in the 'cold hard world' of computing, with its well-defined concepts and deterministic programs.

Du Boulay (1986) 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*". Indeed, it is very plausible that non-viable mental models arise out of students' implicit assumptions about the machine that their programs are supposed to instruct. A notional machine for object-oriented programming is significantly more complex than one for procedural programming (Sajaniemi & Kuittinen 2007). While object-oriented programming is routinely taught to beginners, not all aspects of the notional machine are always adequately addressed in teaching. Pro-

gramming instructors often face the task of helping students construct viable models of the notional machine and steering them clear of non-viable ones. In order to do this, 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 will not attempt to tackle the entire object-oriented notional machine or students' understandings of it. Instead, it will explore a particular subquestion as described below.

### 1.2 Research Question

During Spring 2007, we initiated an explorative study of how introductory programming students view program execution, the notional machine, and computer memory. The research presented in this paper is part of this project, and focuses on a specific, narrow research question, namely:

In what qualitatively different ways do introductory-level students understand the idea of an object stored in memory?

In other words, the point of interest is what students think happens when an object is created and 'placed in the computer'. What is it that exists within the computer after such an operation? This research does not focus exclusively on correct understandings of the phenomenon or on unviable ones (misconceptions), but will discuss them both. As indicated by the research question, the point of view is qualitative, and the aim is to discover and enumerate different understandings, not to assess how common particular understandings may be.

This paper is structured as follows. The next section introduces some related work both on misconceptions and partial understandings of object-oriented concepts and on the phenomenographic research approach used in the present research. The methods used for collecting and analyzing data are described in Section 3. Section 4 presents the results and Section 5 discusses their implications as well as the validity of the research. Section 6 concludes the paper and looks at directions for future work.

## 2 Related Work

### 2.1 Misconceptions and Partial Understandings

Several studies have been reported that explore the ways novice programmers misunderstand concepts in object-oriented programming.

Holland et al. (1997) noted several misconceptions introductory students have about objects. For instance, students may conflate the concepts of object and class, or they may mistake an instance variable storing a name string for an object id. Holland et al. discuss the possible sources for these misconceptions and suggest potential pedagogical solutions. This work is based on anecdotal but intuitively appealing evidence gathered while developing distance education courses.

Fleury (2000) interviewed students on an introductory programming course and found that students form their own rules of what happens and what works in Java programming. For instance, some students thought that the dot operator could only be applied to methods and that the only purpose of a constructor was to initialize instance variables.

Ragonis & Ben-Ari (2005) 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 a wide variety of difficulties students have with object-oriented concepts. Examples of misconceptions found are: “you can write methods that add attributes to classes”; “constructors can only be used to initialize instance variables”; “objects can’t be the values of attributes”; “each method can only be invoked once”. The authors provide a systematic categorization of object-oriented concepts and related misconceptions and other learning difficulties, and offer pedagogical advice for dealing with them.

Teif & Hazzan (2006) observed students of introductory programming on two high school courses and discuss the conceptual confusion of the students in light of partonomic and taxonomic concept hierarchies. For instance, students may incorrectly think that the relationship between a class and objects of that class is partonomic, i.e., that objects are parts of a class.

Instead of investigating misconceptions - incorrect, non-viable ways of understanding a concept - Eckerdal studied the correct but partial conceptions that students have of two key object-oriented concepts: objects and classes (Eckerdal & Thuné 2005, Eckerdal 2006). Her study was based on the phenomenographic research approach and on in-depth interviews, which were analyzed qualitatively. This resulted in a hierarchy of linked understandings of increasing sophistication.

While the studies described above have uncovered various misconceptions and partial understandings related to objects, none of them explicitly addresses the notional machine or explores in detail students’ understandings of storing objects in memory.

## 2.2 Phenomenography and Variation Theory

*Phenomenography* (Marton & Booth 1997) 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 computer science education research, phenomenography has been applied, among other things, to studying beginners’ understandings of programming in general (Booth 1992, Bruce et al. 2004, Eckerdal & Berglund 2005, Stamouli & Huggard 2006) and students’ understandings of specific computing concepts (Eckerdal & Thuné 2005, Berglund 2005, Cope 2000). It has also been used to investigate how the academic community experiences teaching (Lister, Box, Morrison, Tenenberg & Westbrook 2004, Lister et al. 2007) and debates controversial issues (Lister et al. 2006).

To a phenomenographer, learning is characterized by the learner discerning new *dimensions of variation* (Marton & Tsui 2004). Each critical aspect of a phenomenon, i.e., 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. For instance, seeing dogs of different breeds leads to discerning that breed is a critical aspect of dogs. Discerning relationships between values in a dimension and between dimensions of variation leads to still deeper

understanding. To take an example from computer science, Eckerdal & Thuné (2005) concluded that students’ failure to reach more 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 something that act within programs, would need to be shown different examples of relationships between a class description, object actions and resulting events in program execution. In her work on misconceptions mentioned above, Fleury makes the interesting observation that many ‘rules’ students construct about programming are essentially more limited versions of the viable rules we would like them to learn: “the removal of ‘only’ [...] from each of the student constructed rules makes it a true statement” (Fleury 2000). A variation theorist would attribute such misconstrued rules to a lack of perceived variation, which limits the student’s ability to understand the phenomenon in question.

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*. The intention is not to point out which specific kinds of understanding each individual has, but to identify different ways in which the phenomenon can be understood or experienced on a collective level. These categories of description, which arise from the data, represent partial ways of understanding the phenomenon, are logically connected to each other, and are often presented in the form of a hierarchy or tree. An individual person may understand a phenomenon in a number of the different ways represented by the categories of description.

Phenomenography was originally developed outside the ‘hard sciences’, and it seems natural that it traditionally does not problematize the ‘correctness’ of understandings. Many phenomenographic studies aim to discover an array of different, perhaps unorthodox, but nonetheless valid perceptions of a phenomenon. If you investigate people’s perceptions of, say, love, discussing the ‘correctness’ of those perceptions is a lost cause. However, in hard sciences such as computer science, there are concepts that are relatively well defined, and we can say that some understandings are correct and others incorrect. Outcome spaces in phenomenographic projects that deal with such concepts tend to include only understandings that are ‘correct’ in the sense that the researcher deems them to be ‘true statements’ about the phenomenon, i.e., understandings that conform to the intended learning outcome. Wholly, or even partially, incorrect understandings (‘misconceptions’) seem to be commonly discarded by phenomenographers investigating learners’ understandings of computer science concepts. The incorrect understandings are considered not to be related to the phenomenon under study but rather to some other, perhaps imaginary, phenomenon. For instance, an incorrect understanding in which objects are considered parts of a class could be discarded from the outcome space.

Even where it is possible to distinguish between correct and incorrect understandings, concentrating only on the correct understandings is both enough and convenient for some phenomenographic projects, and allows the researcher to delimit the phenomenon more cleanly and to produce neater hierarchies that are arguably easier to report, understand, and make use of. Nevertheless, in computer science education the question of how people *misunderstand* phenomena is interesting and pedagogi-

cally relevant, and the kind of in-depth interviews suitable for phenomenographic analysis often produce intriguing data about the incorrect understandings that learners have. It is this author's impression that the research community is divided in terms of how it views the compatibility of the phenomenographic research approach with studies of incorrect understandings. At one extreme, there are those who argue that incorrect understandings can and surely should be included in phenomenographic results; in fact, making judgements on the correctness of people's understandings is questionable within the phenomenographic framework. An opposite take on the matter is that including incorrect understandings in outcome spaces compromises both the validity of phenomenographic research and the usefulness of the results, which are unlikely to cover very many of the plethora of different misconceptions that people have. This author finds that the prospect of including incorrect understandings in phenomenographic outcome spaces is a promising (yet not unproblematic) one, and will explore it in this paper.

### 3 Research Setting and Methods

The results presented in this paper on students' understandings of storing objects arose as a part of a wider project that investigates students' understandings of memory and program execution. As this research project explores relationships between people and various phenomena, and aims for qualitative results, phenomenography naturally suggested itself as a research approach. This choice of research approach was also influenced by personal circumstances and the proximity of experts in phenomenography.

Following phenomenographic tradition, the advice of experts, and personal intuition, semi-structured interviews (Kvale 1996) were chosen as the data collection method. The following subsections describe the selection of interviewees, the structure of the interview sessions, and the methods used for analyzing data.

#### 3.1 The Students

The interview subjects were from a semester-long university course in introductory programming. 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 drawings in 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 directly participate in running this course.

Approximately one-third through the semester, all students participating in the course were required to complete an online questionnaire about their programming background prior to the course they were taking now and about their attitudes, experiences and workload during the course. Of the several hundreds of respondents, a small subset was selected and invited for interviews based on this background questionnaire. In order to capture a wide range of qualitatively different understandings, the interviewees were hand-picked 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.

Initially, 14 invitations were sent to the selected students via email. Each student was promised two movie tickets as a reward for a one-hour interview. The invitations stressed that the interviews were not a part of the course and would not affect grading in any way. Nine of the students agreed to take part, but one later cancelled

due to scheduling problems. Two more students were selected and sent invitations; both agreed to take part, and the target total of 10 interviewees was reached.

#### 3.2 The Interviews

The author of this paper interviewed the students approximately half-way through the programming course the students were taking. The interviews were recorded in audio. 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 discussion of what program execution means in general. The bulk of the interviews was spent discussing more specific issues related to program execution, computer memory, method invocation, etc. This discussion was organized around two example Java classes. First the students were shown a relatively simple class representing elevators. Where time allowed, a composite class representing buildings that contain elevators was also discussed. Each class also had a main method and formed an executable program. The students were shown the code on paper and asked to describe what they saw there. Unless they spontaneously described what happens when the given programs are executed, they were prompted to do so.

During each one-hour interview, a number of programming topics arose. Not quite the same set of topics was discussed with each student. However, there were a number of focal topics that the interviewer introduced in each interview. One of these, of interest in this paper, was the act of storing 'objects' (though students did not necessarily use that term) in the computer. When the subject of creating objects came up, each interviewee was asked to elaborate on this in more detail with questions such as: What does it mean to have an object created? Does the object exist after it is created? What does the computer know about an object that it has created?

After each interview, the interviewer wrote down some early thoughts about what had been said and about the atmosphere of the interview. In some cases, a brief teaching session also took place, as the interviewer could not bear to let the student leave the room while harboring such misconceptions.

#### 3.3 Data Analysis

The interviewer transcribed each interview and added it to a pool of data, which was analyzed with the different understandings of the collective in mind. The transcription and analysis process started right after the first interview, so that the analysis could provide feedback and ideas to the rest of the interviews.

The data was analyzed with a phenomenographic mindset, with the goal of forming an outcome space of qualitatively different categories of description. The initial analysis was done by the author, then given for a second opinion to a colleague who had also read the transcripts. The resulting discussion brought about some refinement in the category definitions, but no radical changes.

During analysis, it turned out that the interviewed students displayed not only correct understandings of storing objects, but also a number of interesting but incorrect understandings. While a typical phenomenographic analysis (at least in the field of computer science education) might have discarded patently incorrect elements from the data, we chose to include them inasmuch as they had something to do with our phenomenon of interest, storing objects in memory.

### 4 Results

As a result of the analysis, an outcome space with five categories of description was formed. The categories of

Table 1: Overview of Categories of Description

Category	Storing an object is understood as being related to...
PROPERTIES	type-specific properties of some kind.
INSTANCE VARS VARNAME	the object's instance variables. the name of the variable that the object was most recently assigned to.
CODE PARAMETERS	a class's program code. the constructor parameters given upon object creation.

description are labeled according to the main focus of awareness in each category: PROPERTIES, INSTANCE VARS, VARNAME, CODE, PARAMETERS. Table 1 gives an overview of the categories, which are described in more detail and illustrated with quotes in the subsections that follow.

#### 4.1 Category: PROPERTIES

*Storing an object means storing a chunk consisting of object properties of some kind, as defined by the object's type.*

This category describes a general idea that objects are stored within the computer. Each object is defined by some kind of properties, which describe what the object is like. The computer remembers each object's properties somehow in order to make use of them later. Anne<sup>1</sup> observes:

**Interviewer:** So an object is created? How do you perceive that?

**Anne:** Well, it creates like... a chunk or thing which has some properties.

The focus in this category is on objects as multiple composite chunks of data stored in the computer. Objects are not perceived as identical; rather, different objects may have different properties. Distinct objects with different properties constitute values along this vaguely discerned dimension of variation, but it may be unclear exactly what causes an object to have certain properties or what exactly constitutes an object property. How or where objects are stored can also be poorly understood, as illustrated by Anne's response when queried where she thinks objects are after they are created.

**Anne:** Well, it's some unit of information... a part of the machine or... not a part but...

[pause]

**Interviewer:** Where is the object then, or...?

[pause]

**Interviewer:** If it's created, then it exists somewhere somehow?

**Anne:** Yeah, well, somehow within the program, within the computer.

Another dimension of variation is discerned that pertains to object types. Objects of different types can exist and it is important for the computer to know what type each object is in order to determine what the object is like and how it behaves. The type of an object is linked to what kind of properties are stored for it and how the object can be used.

Practically all of the interviewees seemed to consider an object's type to be something that the computer knows for each object, and made implicit use of this fact when describing program behaviour and what properties are stored

for each object. However, few if any put this notion explicitly into words. Ian perhaps comes the closest – he first notes that a newly created object is “an elevator object... an object of the elevator kind”, then explains what happens when a method is invoked on the object.

**Ian:** [The computer] sees that “Gee, this [object] is of the class Elevator”. Then it goes: “What do I do now?”. And then the code says ‘dot getFloor’, so [the computer goes]: “That’s there in the methods [of the class]! Okay.”

All the other, more complex understandings represented by the other categories of description extend this rudimentary type of understanding.

#### 4.2 Category: INSTANCE VARS

*Storing an object means storing the object's state as the values of the object's instance variables.*

This kind of understanding extends the general understanding of object properties described above, and adds a more sophisticated understanding of what it is that defines a particular type of object's properties. A new dimension of variation is discerned that focuses on the values of instance variables. Here is Greg<sup>2</sup> describing what happens when the given elevator class (which has two integer instance variables, `floor` and `topFloor`) is instantiated:

**Greg:** In the main method, it creates an elevator – this `testElevator` – so then it reserves memory slots for the floor and the highest floor, and assigns them values.

Dave explains the contents of an array of elevator objects.

**Interviewer:** You said earlier that there are like many memory slots in the array? So what is in those now when these elevator objects have been placed there?

**Dave:** You mean what there is in one slot, for instance?

**Interviewer:** Yeah.

**Dave:** Well, it holds this elevator object, meaning that it contains all the data that the elevator class contains. It has to have the data showing which floor it's on, and the top floor. Those it has to have at least. So it has two such... ‘bits of information’ in this case. Such integers that it stores for each element.

Brad describes the movement of an elevator as something that changes the state of the elevator object. (The elevator moves via its method `move`, which receives a destination floor as an integer parameter and assigns a new value to `floor` accordingly.)

**Interviewer:** So how do you understand what happens when the elevator is moved?

**Brad:** Well, it has... again, stored in some memory slot the... number where it is at. And in practice, I guess this [assignment statement within the method `move`] means to the computer that a value is assigned again to the memory slot that `floor` refers to. A new value is assigned to the same memory block that used to hold zero, replacing the old one.

Brad's understanding includes a notion of changeable object state, that is, a discernment of variation in how program behaviour is dependent on and affects the stored properties of objects.

<sup>2</sup>Over the course of the interview, Greg expressed multiple different, sometimes contradictory, views about storing objects and program control flow. He seemed to fluctuate between understandings, perhaps indicative of a liminal state (Meyer & Land 2005, Eckerdal et al. 2007).

<sup>1</sup>Interviewee names changed.

### 4.3 Category: VARNAME

*Storing an object means storing the object's state as the values of the object's instance variables, and the name of the variable that the object was most recently assigned to.*

This category, which further extends INSTANCEVARS described above, represents an understanding that focuses on the assignment of objects to variables. Specifically, the name of the variable that an object is assigned to is considered a property of the object, and is stored with the rest of the object's state. A dimension of variation is perceived along which different relationships between objects and variables constitute different values. An object is seen as being different from another if it has been assigned to a differently named variable. As Ian describes a scenario where an elevator object has been created and assigned to a variable named `testElevator`, he first alludes to the elevator being "named":

**Interviewer:** If you think specifically about this scenario with the elevators, what about the elevator is in memory here?

**Ian:** Well, there's precisely that there exists an elevator named `testElevator`. It has been given the value six. I mean, it has been given the value of the top floor which is five and then `this.floor` which is zero at the moment.

Elsewhere in the interview, Ian describes elevators that are stored in an array, which he likens to an Excel worksheet.

**Interviewer:** So in each 'cell' of the 'worksheet' you have...?

**Ian:** The data for one object.

**Interviewer:** So if it's an elevator, that means...?

**Ian:** Well... it would have the `this.floor` of the elevator and the top floor... and then the name, I guess... Yeah.

In the example program elevators are repeatedly added to such an array using the following lines:

```
Elevator newElevator = new Elevator(this.height - 1);
this.elevators[this.numberOfElevators] = newElevator;
this.numberOfElevators++;
```

Ian's description of this code illustrates that 'object names' are not necessarily unique identifiers of objects.

**Interviewer:** You mentioned that a new elevator is created, with the name `newElevator`?

**Ian:** Yeah. So the [two elevator objects so far created and placed in the array] have the same name as a matter of fact.

**Interviewer:** There are several elevators named `newElevator` in the array, or?

**Ian:** Yeah, there are.

Greg is another interviewee who considered variable names to be part of an object's data. For Greg, repeated assignment of an object value involves changing the object's name. Below, Greg examines the statement `Elevator test1 = office.orderElevator(2);`. He has established that the method call finds the closest elevator in a building, moves it to the given floor and returns it. He then explains what happens to the elevator:

**Greg:** I suppose it changes its name to be `test1`, then...

**Interviewer:** Changes the name of what to be `test1`?

**Greg:** The closest elevator which it fetches with this [method call].

**Interviewer:** It renames the memory slot to be `test1`?

**Greg:** Renames the elevator that's in the slot. Since it had been assigned the name `newElevator`, it changes it to `test1`.

**Interviewer:** All right.

**Greg:** And I suppose its floor also changes as it's ordered.

For Ian, reassignment means making a copy of the elevator, only with a different name.

**Ian:** The elevator that has gone there [to the floor that it was ordered to]... It decides that its name is now `test1`.

**Interviewer:** So the name of the elevator changes?

**Ian:** Yeah it changes to `test1`... Mmm, or I mean no... I mean now an elevator named `test1` exists in addition to the original still being there in the array.

### 4.4 Category: CODE

*Storing an object means storing a copy of the program code of the class that defines the object's type.*

This category places a focus on the program code that defines the object's type, its instance variables and methods. This code is perceived as being included in the properties that define an object. Eric has this view:

**Interviewer:** So what is there after it's created it in memory? How do you think about it?

**Eric:** Well, there in memory are at least all these instructions in the code, all these characters and spaces and newlines.

Whenever an object is created, the computer is understood to take the class definition and store a copy of it. A dimension of variation is perceived where distinct copies of code are characteristic of distinct objects. For instance, Greg pointed out that for each element in an elevator array, all the code in the elevator apart from the main methods needs to be stored.

### 4.5 Category: PARAMETERS

*Storing an object means storing the constructor parameters given upon object creation. These parameter values unambiguously define the object's properties.*

This category represents an understanding that focuses on constructor parameters as the defining properties of objects. Not only do an object's creation parameters have an effect on initializing the object, but the object is thought to be stored in memory in terms of those parameters. A dimension of variation is discerned where the objects created with different parameter values constitute examples of different objects in computer memory. Fred shows an example of this kind of understanding:

**Fred:** It remembers what value that particular elevator object has received for the parameter.

**Interviewer:** Which parameter?

**Fred:** (Points at `new Elevator(5)` in the given code.) Like, here where an elevator was created and received five as a parameter.

Fred's understanding of object properties is limited to the **PARAMETERS** category. He does not have a notion of a persistent object state. In keeping with the idea of storing the constructor parameters for each object, he considers an object to "start with a clean slate every time [any of its methods] is called". Whenever an object's method is called, Fred consistently considers its execution to start with the object in whatever state the class's constructor sets for it. For him, method calls do not have a lasting impact on state and even assignments to instance variables last only as long as the current method call.

#### 4.6 Relationships between Categories

In the category **PROPERTIES**, the focus is on objects as 'chunks' of data in the computer. Variation is discerned in how these chunks differ in terms of some type-specific properties. Variation in what different kinds of properties objects have is linked to variation in the objects' types. However, while variation in object properties is discerned, the relationship of those properties to the program that determines the objects is not necessarily discerned at all. Variation in object properties is vaguely if at all linked to program structure (e.g. instance variables). This kind of understanding is viable and not incorrect, but by itself not enough for understanding how the computer handles objects in object-oriented programs. This category serves as a basis for other categories of description, which extend this rudimentary understanding of storing objects and link object properties to specific features of program structure.

The category **INSTANCEVARS** places a focus on the instance variables of an object. They are seen as key features that define what a computer stores about an object. Variation in objects' properties is discerned as being linked to variation in the values of objects' instance variables. This is a viable, more concrete and useful extension of the category **PROPERTIES**.

An understanding of the **VARNAME** variety is characterized by mistaking a critical feature (name) of one concept (variables) for a critical feature of another concept (objects). This is an incorrect 'over-extension' of **INSTANCEVARS** that focuses on assignment operations and links variation in assignments to variation in objects' stored properties. A name that an object has been assigned to is seen as part of the object's state when in fact assignments are irrelevant to object state. This kind of understanding is a symptom, cause, or both, of not being able to distinguish between object-valued variables and object representations in memory. In other words, the category is related to the often-reported inability of students to grasp the idea of references, which are known to be a difficult topic in learning programming (Adcock et al. 2007). Thinking of a variable name as a part of an object's state is likely to result in a non-viable mental model of references. For instance, Ian has a mental model where dot notation means that the computer looks through the contents of memory to find an object with a matching name. This misunderstanding is also related to the 'identity/attribute confusion' reported by Holland et al. (1997) and Ragonis & Ben-Ari (2005).

The category **CODE** includes the entire code of a class within the focus of awareness. This category extends **PROPERTIES**: it encompasses the notion that a computer needs to know each object's type at runtime, and adds a dimension of variation where a separate copy of the same code is attributed to each object of a particular type. According to this kind of understanding, it is storing the code

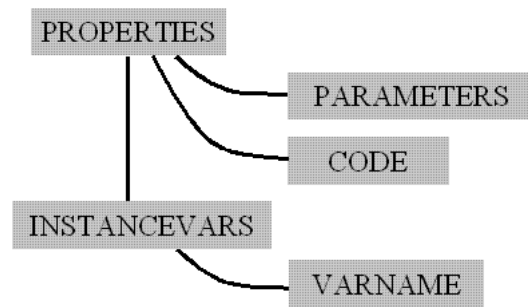


Figure 1: Relationships between categories of description. Each line indicates that the category below encompasses and extends the category above.

that is key to defining what an object is. While the idea of storing each object's runtime type is correct, the notion that each object has a copy of the class's code is not. Still, this misunderstanding – which has some intuitive appeal – may be a viable way to think about the matter in what comes to introductory programming, and may be relatively harmless in terms of CS1 studies. However, having this understanding may also be indicative of difficulties in distinguishing between classes and objects, an often-reported beginner problem.

The category **PARAMETERS** extends and concretizes **PROPERTIES** in yet a different way. The focus in this category is – incorrectly – placed on the values of constructor parameters. A relationship is discerned between variation of construction parameters and variation in what is stored about an object, so that the former fully defines the latter. A person with this kind of understanding is unlikely to form an understanding about object state, and will probably find it relatively hard to see the competing, even contradictory (but correct) idea of storing objects' instance variables. This was certainly the case with Fred, quoted in the previous section.

Table 2 summarizes the five categories of description. The relationships between the categories are illustrated in Figure 1.

Typically, in a phenomenographic outcome space that takes the shape of a tree, there are multiple branches that represent partial but correct understandings, which are extended by the 'root' of the tree, which represents a more complete way of understanding the phenomenon. The reader may note that Figure 1, while tree-shaped, is not typical in this sense. Instead, only **INSTANCEVARS** represents a correct, more complete extension understanding of **PROPERTIES**. It encompasses all the correct understandings represented in the diagram, and in this sense corresponds to the rich 'root' understanding in a more typical outcome space. The other three categories 'branching out' from the two correct categories represent partially incorrect understandings. In Figure 1, there is no one category which would encompass and extend all these branches (since there is no category of understanding which would encompass the various partially incorrect understandings).

## 5 Discussion

### 5.1 On Incorrect Understandings in Phenomenographic Results

Of the five categories of description that emerged from the data, **PROPERTIES** and **INSTANCEVARS** represent viable understandings that we would like our students to have. These understandings are 'true' in the sense that they correspond to the actual technical reality of the computer. To learn object-oriented programming, it is crucial to discern that object types and instance variables are key to how objects exist within the computer. Not all students discern

Table 2: Summary of Categories

Category	Focus	Description
PROPERTIES	object properties	Storing an object means storing a chunk consisting of object properties of some kind, as defined by the object's type.
INSTANCEVARS	instance variables	Storing an object means storing the object's state as the values of the object's instance variables.
VARNAME	assignment to variables	Storing an object means storing the object's state as the values of the object's instance variables, and the name of the variable that the object was most recently assigned to.
CODE	program code	Storing an object means storing a copy of the program code of the class that defines the object's type.
PARAMETERS	constructor parameters	Storing an object means storing the constructor parameters given upon object creation. These parameter values unambiguously define the object's properties.

this link, however, and some are distracted by other features of objects, as illustrated by the other three categories.

In Figure 1, each of the three categories that ‘deviates’ towards the right extends a correct understanding in some incorrect way. That is, these understandings – PARAMETERS, VARNAME, and CODE – are partially incorrect. They represent understandings that do not correspond to the reality of the computer and that are, to varying degrees, harmful for learning programming. A learner with one of these partially incorrect understandings is focusing their awareness on something that is not helpful for reaching the intended learning outcomes. They mistakenly link to the phenomenon of storing objects a dimension of variation that is unwanted from the pedagogical point of view. The problems that a learner faces if they associate incorrect dimensions of variation with a phenomenon may be further compounded by the inability to discern the correct dimensions of variation.

In phenomenography, the researcher, drawing on the data, decides what falls within the scope of a phenomenon. This is to be done in such a way that the categories in the outcome space can be said to describe different understandings of one phenomenon. We find that in our case, it is not only acceptable but in fact important to include both correct and partially incorrect understandings of the phenomenon. Since students’ understandings of storing objects can clearly be incorrect in pedagogically interesting ways, it behoves us to include these incorrect understandings when enumerating qualitatively different ways of understanding the phenomenon. Linking the incorrect understandings to the correct ones seems like a promising way to make sense of them systematically. As discussed in Section 2, this research is not typical phenomenography as it is applied to CS phenomena, and the inclusion of partially incorrect understandings in the outcome space raises some pertinent questions.

There is a limitless number of understandings – and misunderstandings – of each phenomenon. Phenomenography posits that there are not very many qualitatively different understandings. Does this change if we include partially incorrect understandings in the scope of our studies? Are we researching multiple phenomena while pretending to research just one? Are we faced with incomprehensible outcome spaces with countless categories of description?

We argue that the inclusion of *partially* incorrect understandings is a valid application of the phenomenographic research approach. All the understandings reported in this paper share a correct core in that they include an understanding of an object stored in the computer as a collection of properties that depends on the type of the object. This correct core links all the understandings reported above and serves to delimit the phenomenon. Care must be taken to determine that the incorrect understandings pertain to the actual phenomenon under investigation and are not completely unrelated figments of people’s imaginations. This is a challenging task, but not one that is new to phenomenography: the phenomenog-

rapher always has the responsibility delimiting the phenomenon of interest, be it ‘love’, ‘the decision-making process in our company’, or ‘storing objects in the computer’. We argue that the correct part of an outcome space, which represents correct understandings of a phenomenon (e.g. PROPERTIES and INSTANCEVARS), can be used, as above, to help distinguish between partially correct understandings and wholly incorrect, unrelated understandings. This helps delimit the phenomenon and the size of the outcome space even when incorrect understandings are included in a study.

Another issue worth noting is coverage. It seems to be possible, even in a small-scale study, to construct an outcome space with a good coverage of the correct understandings that one can expect to find in a similar group of people. If partially incorrect understandings are included in the study, the size of the task increases, and it is likely that a lesser coverage will be achieved. Nevertheless, such a study can discover valuable qualitative information, contributing towards a body of knowledge that other studies – qualitative and quantitative – can complement. It is also worth noting that not all incorrect understandings are equally common nor pedagogically relevant; the more common ones are likelier to be discovered first.

A future challenge to phenomenography could be to investigate people’s understandings and misunderstandings of the relationships between phenomena. Many beginners have trouble distinguishing phenomena (e.g. object/class, object/variable) from one another, and their understandings of one phenomenon may conflate with their understandings of another. As we explore understandings of one phenomenon, we end up ‘charting the borders’ that the phenomenon has with other phenomena and exploring overlapping aspects of multiple phenomena. This poses challenges to phenomenography, but we feel that further work on the research approach could help produce interesting insights into people’s understandings and misunderstandings.

## 5.2 Pedagogical Implications of the Results

In light of constructivist theory and the results presented above, it seems clear that people construct a number of different interpretations of how a computer keeps track of object data. Especially when little explicit attention has been paid to the notional machine in teaching, as was the case on the CS1 course we investigated, students are liable to come up with their own models of it. Sometimes they get it right, sometimes they don’t. From our data, it is obvious that some students had constructed incorrect models of storing objects in the computer.

The results presented in this paper highlight two kinds of learning difficulties. On one hand, some students are not able to focus on all the important aspects of storing objects and do not link the phenomenon to the variation in the values of instance variables as we would like them

to. As programming instructors, we need to draw students' attention to the important aspects and variation in those aspects, and underline their importance where possible. Visualizations in course materials or visualization tools – e.g. Jeliot (Moreno et al. 2004) or the BlueJ object bench (Kölling et al. 2003) – can help here by making explicit and visible the idea that objects' states are defined by those objects' instance variables.

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. For example, Fred from Subsection 4.5 would have benefited from teaching that draws his attention to the variation in the values of instance variables and how this links to the behaviour of objects, while at the same time underlining the lack of variation with respect to constructor parameters. It seems a good idea to show him an example with multiple objects of the same type that end up being different despite having been constructed with the exact same constructor parameter values. Using variation of a critical aspect in combination with the intentional lack of variation in another should be a useful tool in dispelling incorrect understandings. Fred might also learn better from seeing and interacting with visualizations of changing object states. (See the work of Yehezkel et al. (2005) for one example of using a visualization tool to prevent the rise of misconceptions.) Similarly, careful misconception-aware instruction and visualizations could also help dispel the problems represented by the category VARNAME by clarifying object-variable relationships to students.

As noted above, an understanding described by the category CODE may – despite being untrue – be relatively harmless for a beginner. Even so, showing in a concrete way how classes exist as separate entities in the machine and how each object can be stored without having to store a copy of the code should help towards understanding the difference between classes and objects.

In summary, we think that the use of variation of the correct critical aspects, as suggested by variation theory, is an excellent pedagogical tool. However, the impact of any pedagogical method will vary depending on each student's prior mental model, which is why it is very valuable for us teachers to know about the specific pitfalls (i.e., incorrect understandings) that students may be in. To help our students avoid those pitfalls, to recognize when they fall in them, and to help them out, we should be aware of at least the most common incorrect understandings related to the phenomena we teach. By recognizing what critical aspects students most often incorrectly focus on, we can make more efficient use of the pedagogical advice provided by variation theory and be better prepared to help our students construct viable models.

## 6 Conclusions and Future Work

This paper has reported a phenomenographic study of how introductory students view objects that have been created and stored by an object-oriented program. The results suggest that students come up with many kinds of understandings of this phenomenon, not all of which are correct. While the qualitative results presented in this paper do not give an idea of how widespread particular misunderstandings are, they do paint a somewhat worrying picture by illustrating a number of ways in which students misunderstand some of the most fundamental object-oriented concepts. This is unsurprising: these results complement earlier findings, which suggest that learning programming in general is very troublesome and difficult to many students (McCracken et al. 2001, Lister, Seppälä, Simon, Thomas, Adams, Fitzgerald, Fone, Hamer, Lindholm, McCartney, Moström & Sanders 2004). It is also not unprecedented that students struggle with and misunderstand the basic concepts of a field of study, as shown by physics stu-

dents' difficulties with understanding the concept of force, for instance (Hestenes et al. 1992). Teachers of programming should be aware of and look out for such misunderstandings in order to facilitate the construction of correct, viable models.

This paper contributes to computer science education research in two different ways. First, it has presented detailed results of an in-depth study of a students' understandings of a particular phenomenon, storing objects. These results are concretized and illustrated with quotes, and highlight some pedagogically relevant misunderstandings. Second, the paper serves as an example of methodologically exploratory research, which brings together two previously disjointed branches of computer science education research: the study of misconceptions and the phenomenographic research approach. The phenomenographic approach used in this study extends traditional phenomenography by including partially incorrect understandings in a phenomenographic outcome space, and explicitly treating them as such. This approach offers a new way of studying misconceptions and linking them to dimensions of variation and to correct understandings of the phenomenon.

There are many paths for future work that can follow from the present study. The effects of visualizing the notional machine on students' understandings of storing objects could be assessed. Other aspects of storing objects could be explored: for instance, this study has examined *what* students think objects are, but has largely ignored the issues of *where* or *how* the computer stores objects. Still other aspects of the notional machine could be explored, e.g. the structure of memory, the management of control flow in programs, and so forth. The possible relationships between students' learning goals and field of study and the resulting understandings could be charted out.

Last but not least, the prospect of investigating misunderstandings using phenomenography seems to have potential, but needs further theoretical work. A vocabulary for discussing partially incorrect understandings in phenomenographic outcome spaces is needed, and there are many open questions. What is the best way to deal with a slew of different incorrect understandings? Exactly what kind of criteria are used to delimit a phenomenon? What exactly is the nature of the links between correct understandings and partially incorrect ones? How should incorrect understandings be reported so that they can be of use? What is the stage of a phenomenography-based research project at which it is appropriate to make judgements about the correctness of people's understandings? Can we use phenomenography to explore people's understandings and misunderstandings of relationships between two or more phenomena? Hopefully, this paper can serve as a basis for discussing some of these interesting issues.

## Acknowledgements

My thanks to Anders Berglund and Lauri Malmi for many fruitful discussions about the nature of phenomenography and for their comments on this paper. Thanks are also due to the staff of the CS1 course for their help in organizing the programming background and attitudes questionnaire and for letting me interview their students.

## References

- Adcock, B., Bucci, P., Heym, W. D., Hollingsworth, J. E., Long, T. & Weide, B. W. (2007), 'Which pointer errors do students make?', *SIGCSE Bulletin* **39**(1), 9–13.
- Ben-Ari, M. (2001), 'Constructivism in computer science education', *Journal of Computers in Mathematics and Science Teaching* **20**(1), 45–73.

- Berglund, A. (2005), 'Learning computer systems in a distributed project course: The what, why, how and where'.
- Booth, S. (1992), Learning to program: A phenomenographic perspective, Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden.
- Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M. & Stoodley, I. (2004), 'Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university', *Journal of Information Technology Education* **3**, 143–160.
- Cope, C. J. (2000), 'Educationally critical aspects of the experience of learning about the concept of an information system (PhD Thesis)', <http://ironbark.bendigo.latrobe.edu.au/~cope/cope-thesis.pdf>.
- Du Boulay, B. (1986), 'Some difficulties of learning to program', *Journal of Educational Computing Research* **2**(1), 57–73.
- Eckerdal, A. (2006), 'Novice students' learning of object-oriented programming', Licentiate thesis.
- Eckerdal, A. & Berglund, A. (2005), What does it take to learn 'programming thinking'?, in 'Proceedings of The First International Computing Education Research Workshop', pp. 135–143.
- Eckerdal, A., McCartney, R., Mostrom, J. E., Sanders, K., Thomas, L. & Zander, C. (2007), From limen to lumen: Computing students in liminal spaces, in 'Proceedings of ICER'2007, International Conference of Computing Education Research', ACM Press, New York, p. [forthcoming].
- Eckerdal, A. & Thuné, M. (2005), 'Novice Java programmers' conceptions of "object" and "class", and variation theory', *SIGCSE Bulletin* **37**(3), 89–93.
- Fleury, A. E. (2000), 'Programming in java: student-constructed rules', *SIGCSE Bulletin* **32**(1), 197–201.
- Hestenes, D., Wells, M. & Swackhamer, G. (1992), 'Force concept inventory', *The Physics Teacher* **30**, 141–158.
- Holland, S., Griffiths, R. & Woodman, M. (1997), 'Avoiding object misconceptions', *SIGCSE Bulletin* **29**(1), 131–134.
- Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (2003), 'The BlueJ system and its pedagogy', *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology* **13**(4).
- Kvale, S. (1996), *InterViews: An Introduction to Qualitative Research Interviewing*, Sage Publications.
- Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., Bower, M., Carbone, A., Davey, B., de Raadt, M., Doyle, B., Fitzgerald, S., Mannila, L., Kutay, C., Peltomäki, M., Sheard, J., Simon, Sutton, K., Traynor, D., Tutty, J. & Venables, A. (2007), Differing ways that computing academics understand teaching, in 'ACE '07: Proceedings of the ninth Australasian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, pp. 97–106.
- Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C. & Whalley, J. L. (2006), 'Research perspectives on the objects-early debate', *SIGCSE Bulletin* **38**(4), 146–165.
- Lister, R., Box, I., Morrison, B., Tenenber, J. & Westbrook, D. S. (2004), 'The dimensions of variation in the teaching of data structures', *SIGCSE Bulletin* **36**(3), 92–96.
- Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E. & Sanders, K. (2004), 'A multi-national study of reading and tracing skills in novice programmers', *SIGCSE Bulletin* **36**(4), 119–150.
- Marton, F. & Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associates.
- Marton, F. & Tsui, A. (2004), *Classroom Discourse and the Space of Learning*, Lawrence Erlbaum Associates.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A multi-national, multi-institutional study of assessment of programming skills of first-year CS students', *SIGCSE Bulletin* **33**(4), 125–180.
- Meyer, J. H. & Land, R. (2005), 'Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning', *Higher Education* **49**, 373–388.
- Moreno, A., Myller, N., Sutinen, E. & Ben-Ari, M. (2004), Visualizing programs with Jeliot 3, in 'Proceedings of the International Working Conference on Advanced Visual Interfaces', Gallipoli (Lecce), Italy, pp. 373 – 376.
- Ragonis, N. & Ben-Ari, M. (2005), 'A long-term investigation of the comprehension of OOP concepts by novices.', *Computer Science Education* **15**(3), 203 – 221.
- Sajaniemi, J. & Kuittinen, M. (2007), From procedures to objects: What have we (not) done?, in J. Sajaniemi, M. Tukiainen, R. Bednarik & S. Nevalainen, eds, 'Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group', University of Joensuu, Department of Computer Science and Statistics, pp. 86–100.
- Stamouli, I. & Huggard, M. (2006), Object oriented programming and program correctness: the students' perspective, in 'ICER '06: Proceedings of the 2006 international workshop on Computing education research', ACM Press, New York, NY, USA, pp. 109–118.
- Teif, M. & Hazzan, O. (2006), 'Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts', *SIGCSE Bulletin* **38**(4), 55–60.
- Yehezkel, C., Ben-Ari, M. & Dreyfus, T. (2005), Computer architecture and mental models, Vol. 37, ACM Press, New York, NY, USA, pp. 101–105.



# Computer Science Students' Experiences of Decision Making in Project Groups

Mattias Wiggberg

Dept. of Information Technology, UpCERG  
Uppsala University,  
Box 337, S-751 05, Uppsala, Sweden,  
Email: Mattias.Wiggberg@it.uu.se

## Abstract

This paper describes a study intended to understand the ways in which students experience the process of decision-making in computer science student projects. It also investigates the ways the student team works to make decisions.

The empirical setting for the study is a semester-long project with 22 final year computer science students. It is a qualitative study where data are gathered using interviews and analyzed using phenomenography.

Six categories have been identified describing how students experience the process of decision-making in computer science projects. The level of sophistication differs between the categories. The first describes an experience of decision-making as individual decisions too small and unimportant to be handled by anyone other than the individual. At the other end is the experience of decision-making as a democratic process involving both the full group and the context in which the group acts. The other four categories are situated between these two extremes.

## 1 Introduction

Learning in computer science project courses is affected by, among other things, how the project is structured. Structures for decision-making among the students in the projects are thus important in order to design good learning environments. According to Desanctis & Gallupe (1987) a decision making team is

[...]two or more people who are jointly responsible for detecting a problem, elaborating on the nature of the problem, generating possible solutions, evaluating potential solutions, or formulating strategies for implementing solutions. (Desanctis & Gallupe 1987, p. 590)

The perspective of the student project team as a decision-making team brings new questions to the fore. In this context, it is relevant to consider how students experience decision-making.

Largely, universities today organize education so that teamwork becomes an integral part of students' education. In computer science, this is manifested in the important role of teamwork in the ACM Curriculum (The Joint Task Force for Computing Curricula

2005), as well as in many study programs. The Masters program in Information Technology at Uppsala University is one example where projects are emphasized as a model for learning approaches.

The study reported here on decision-making experiences is related to previous studies, e.g. on power structures (Wiggberg 2007). The intention is that the results from the various studies will be combined to form a cohesive contribution to the area of project approaches in computer science education. Regardless of the differing focuses, the unit of analysis in all is the collaborating student project team.

The students' own experience of their learning related to the aforementioned issues is explored with qualitative data analysis. The research framework in the current study, as in the initial study, is built on phenomenography. Marton and Booth (Marton & Booth 1997) provide a general discussion of phenomenography, and Berglund (Berglund 2005) of its applications within computer science education.

Learning outcome is defined by Berglund (2005):

The learning outcome that is sought is that which is actually learned from the point of view of what is meant to be learnt. (Berglund 2005, p. 39)

Exploring the relationship between learning outcomes and decision mechanisms in computer science project courses can help us to understand different outcomes of learning. Knowledge about this relationship will give new and potentially valuable clues in the field of learning outcomes in software teams. A possible contribution to practitioners in the field of computer science education is a set of guidelines for defining and communicating learning purposes in practitioners' teaching.

### 1.1 Research Questions

Investigating the distribution of power in a computer science student project (Wiggberg 2007), it is clear that decision-making is a visible structure that determines much of the work in projects. That is also the rationale for studying the structure of decision-making mechanisms in the student teams.

By talking to students and reading through interviews from previous research, two different research questions became apparent. The first one, the primary, can be called "decision-making processes" and concerns

- how the student experiences the process of decision-making.

This question was the driving force in the study. During the investigation, a second question became apparent:

- in which ways does the student team work to make decisions?

The two research questions differ not only in their content but also in their perspectives. The first question is about a student experience, while the second concerns a structure for decision-making. The decision-making processes question is about the student experience of a certain phenomenon at a collective level. The question of structures for decision-making regards the manner, or the structure, of the process of decision-making. This question does not concern student experiences as such, but rather looks at the system within which people experience things. These two questions therefore have different perspectives and require different approaches. The first question asks for answers from the student's perspective, while the second asks for the perspective of an outsider observing the students. These differences led to the use of different methods for analyzing the data. For the first question, we analyzed the data using phenomenography; for the second, we simply categorized and summarized our findings. A discussion on these two follows in section 4.

To address the first question we required a research framework that helps the researcher understand the experience of the student. Phenomenography is a second-order research perspective: it tells the researcher something about other people's experience of the world, whereas a first-order research perspective makes statements about the world itself (Marton & Booth 1997). Thus phenomenography was chosen as research framework to explore how students experience the process of decision-making. The second question, on the other hand, requires a first-order research perspective.

Another advantage of phenomenography is that it aims to gain knowledge on the collective level. The individual experience is important, but only as part of the whole student cohort. Regardless of how the students have divided themselves within the project, for this study they constitute a single data set.

This study is restricted to learning about the decision-making experiences and structures identified by the students in the project. It does not focus on such questions as why the decision-making structure looks as it does or why the experiences came about.

## 2 Related Work

Related work in connection to the research questions considers both theory on structures for decision-making and research in educational settings for project work. Firstly, a walk through of some of the major theoretical and analytical views on decision-making structures will be done. Secondly, studies on projects as educational settings within or close to computer science will be presented.

### 2.1 Decision-Making Structures

Organizational decision-making is a complex process involving several different steps. The rational decision-making model divides the process into six analytical steps: identify the problem to be solved; choose the best decision style; develop alternative solutions; choose among the solutions developed; implement the selected alternative; and evaluate the effect of the choice. It is important to notice that the rational model does not support how people and organizations make decisions, but gives the analytical frame for analysis of such decision-making (McShane & Glinow 2005). The rational decision-making model is by no means universally accepted; see, for example,

Simon (1955). In this paper, the analysis of decision-making will be limited to steps three and four, developing and choosing between solutions.

Barker et al. (1991) suggest five strategies for team decision-making: force, majority vote, compromise, arbitration, and consensus. No single strategy is thought to be best for all teams. Instead the most appropriate team decision-making strategy is likely to depend on the particular group phase, time constraints, and other such factors (Barker et al. 1991).

Wickens & Hollands (2000) extend the discussion on decision-making with domains of decision-making, a model proposed by Shanteau (1992). In this model, the value of practice and experience in a field is questioned in certain domains. Einhorn & Hogarth (1978) have then added understanding of feedback to the model. The model presents the characteristics of good and poor decision-making domains. A good domain is dynamic, involves decisions about things, has decomposable decision problems, and provides feedback. A poor domain is static, involves decisions about people, gives no or poor feedback, and does not provide decomposable decision problems (Shanteau 1992). The conclusion is that in poor domains, decision-making is hard even for experienced people.

Wickens & Hollands (2000) note that an important component of effective decision-making is situation awareness, the understanding of the situation, often by diagnosing which possible state the world is in (Swets & Pickett 1982).

McShane & Glinow (2005) reason on effective team decision-making and state that in many situations teams potentially make better decisions than individuals. However, many group mechanisms can impair the effectiveness of the group. Janis (1989) suggests that no team member, including the leader, should be too strong or influential. Fiest (1997) points out the importance of keeping the team size on a moderate level: big enough to do necessary work and small enough not to consume too many resources.

### 2.2 Projects as an Educational Setting

The issue of project work as an educational setting in engineering and computer science has been investigated in several papers, for instance Brown & Dobbie (1998), Coppit & Haddox-Schatz (2005), Newman et al. (2003) and Leeper (1989).

Seat & Lord (1998) emphasize the importance of practising interpersonal skills such as communication and teaming. They refer to a program for teaching interaction skills to engineering students with the aim of increasing the efficiency of their technical skills. The approach for teaching those soft skills was to let the students adopt a simple set of general principles and apply them to their own context. From there, the students could experiment and interact in supervised groups with the possibility of getting feedback.

Berglund (2005) explores students' learning within a similar project environment. Among other things, Berglund identifies three different motives for taking the course in focus: academic achievement, project and team working capacity, and social competence.

Barker (2005) reports on how perceived pressure to finish a project for clients, together with poor understanding of how to work well in groups, has a negative impact on the learning environment and pedagogic outcome of the project model.

When students are allowed to select their roles based on expediency or comfort, it works against the benefits of collaborative learning, particularly in the case of IT education. While this approach may seem eminently practical and efficient, it does not

provide any of the students with a new learning experience, but instead practice of existing skills. (Barker 2005, p. 279)

Hence, when students select their own roles within the team, they tend to choose tasks for which they already have well-developed skills, and through that choice lose the major impact of the peer learning exchange expected in collaborative work. Barker also argues that only when group processes are made explicit can activities lead to enhanced learning. Even though performed in different cultural and social context from the current project, Barker's work presents findings worth considering.

An earlier study investigated a full-semester engineering project course at the Department of Information Technology, Uppsala University, with many similarities to the current course. The focus of the study was on how power is distributed within a group of students. The students taking the course were in their final year of the IT engineering program. The course duration was one semester (Wiggberg 2007). The students worked on the task of designing and building a power line inspection robot (Danielsson et al. 2006). The study showed, as expected when it comes to expert power (Raven & French 1960), that computer science skills are shown to be a contributing factor when it comes to power within the group. Finally, three qualitatively different ways of experiencing other students' computer science skills are revealed: by presumed skills, by earlier demonstrated skills, and by skills demonstrated in the actual project.

Waite et al. (2004) have reported from a study of computer science students in undergraduate project courses where there are indications that the students perform poorly in group skills. By ethnographic observation and in-depth interviews of students during projects, they attempted to discover why using the project model did not give the students these skills. They state:

In order to improve the students' collaborative skills, we need to change some of the characteristics of their occupational community. This cannot be done by teaching a course in group work or telling them to work in groups to solve a problem. It has to be done by understanding the enculturation process, and establishing conditions that favour development of a collaborative culture. (Waite et al. 2004, p. 14)

Waite et al. emphasize the importance of not just adopting the project model, but instead carefully designing the project course in order to achieve the desired learning outcome.

The same study concludes that group decision-making is often experienced as an ineffective and time-consuming process. Two characteristics of the decision-making process contribute to this: team members' predilection for their own opinions and their distrust in the rationality of using decision-making methods. By experimentation, the authors developed a viable group decision-making exercise that helps students to retreat from favoring the individual choice in decision-making situations (Waite et al. 2004).

Entwistle (1977) discusses the need for reflection on group methods and points at the importance of group methods in higher education:

What may, however, be necessary is to think more clearly about the functions of large-group and small-group methods in relation to the particular intellectual skills, or cognitive style, they are expected to foster and

whether the assignments and examination questions given to students provide sufficient encouragement for deep-level processing. (Entwistle 1977, p. 235)

Even though computer science project teams have been researched in recent years, there is still a gap in the knowledge of the impact of decision-making. This study can therefore, among other things, contribute to the body of research on the learning process within computer science student projects with information on how students experience decision-making. By revealing this information, we can learn more about one of the factors in project structures.

### 3 The Setting

The computer science project course studied was held in the final year of the Computer Science Masters program at the Department of Information Technology, Uppsala University, between August 2006 and January 2007. The course was taught in English.

The course duration was 20 weeks, 10 weeks part time in parallel with another more traditional course and 10 weeks full time.

Participating students work with one project for the full duration of the course. The product, which varies somewhat by course instance, is not specified with any exactness. Instead, the students are expected to formulate the requirement specification themselves from an initial idea proposed by the team of teachers in cooperation with the participating industry partner. Students do not need to complete the product in order to pass the course, since the focal point is the process of working on the product. This project falls within the framework of Open Ended Group Project courses (OEGP) (Faulkner et al. 2006).

The number of distinct projects varies with the number of students. In the current course, 22 students participated and were divided in two project teams where they carried out either (1) a task involving designing the software for a game for cell phones (Nilsson et al. 2007) or (2) a cell phone positioning task (Back et al. 2007). The industrial partners also contributed to the project as mock customers.

Essentially, the same course has been run for over twenty years. The tasks have varied greatly. Examples from the past five years include football robots, map-making systems, real-time middleware for robots, distributed mobile games, and GPS systems (Pettersson 2006). Daniels & Asplund (2000) and Wiggberg (2007) have described earlier instances of this course.

#### 3.1 The Physical Environment

The physical environment of the project plays an important role in a project (Jaques 1995, p. 120). During the project the students worked in two project rooms. Each team sat in a separate room, but the rooms were located close together. Collaboration between the project teams was encouraged. The work environment was an open-plan office where people located themselves close to the other members of the smaller groups they ended up working in. Each student was given a workspace and a computer. The room was equipped with whiteboard, printer, and some other hardware. The teams were asked to use software for keeping track of bugs, version handler, content management system and personal diary software (Pettersson et al. 2006).

The students were expected to work 8 hours a day during the second half of the semester, and presence

was compulsory from 9 am to 4 pm (Pettersson et al. 2006).

### 3.2 Project Teams and Their Tasks

Twenty-two students participated in the course. Five of them were exchange students from Tongji University, Shanghai, China, who had completed two years of computer science in China and one year at the department prior to the project course.

The other 17 students were Swedish and were all enrolled in the Computer Science Masters Program. They had completed approximately two years of compulsory courses and one year based on individual preferences. Both the exchange students and the Swedish students voluntarily applied for the course <sup>1</sup>.

Two different project teams, with different tasks, were formed at the beginning of the project course. Despite the difference in tasks, there was a high level of collaboration between the teams.

The project team Point of Interest (POI) was assigned the overall task of designing and implementing a mobile positioning system based on information provided by the GSM <sup>2</sup> network and GPS <sup>3</sup>/WLAN <sup>4</sup> when available. The second part of the task was to create a map on which points of interest could be displayed (Back et al. 2007).

Project team Teazle Goes Mobile (TGM), was assigned the task of implementing a distributed multiplayer game for mobile devices. The game was originally developed in 1997 and called Teazle (Nilsson 2006).

Both tasks were rather complex and involved a client and server solution as well as a graphical web front end.

Although the technical goals were given by the industry partner and the team of teachers, the specific shape of the technical goals, as well as design and implementation issues, were left to the project teams to decide. Faced with a somewhat vague design, the project teams had to interpret the task and develop a system design, a requirement specification, and an implementation plan.

The team members originally organized themselves around the three major development areas. The TGM areas, based on figure 1, were client side, server side, and web portal. POI organized themselves similarly, although the software components looked slightly different. The server side sub-team, with four members, took care of the login server, the game server, the game database, and the web database. The client side sub-team, with five members, took care of the mobile application. Finally, the web portal sub-team had two people working on the game's web interface.

Project managers for each team were appointed by the teaching team following applications from team members.

Additional responsibilities for all three sub-teams included lead programmer, testing manager, system administrator, configuration manager, bug manager, final report manager, user interface manager, and requirement specification managers (Nilsson 2006).

<sup>1</sup> Anders Berglund, Director of international undergraduate collaboration, Department of Information Technology, Uppsala University, private communication.

<sup>2</sup> Global System for Mobile communications, GSM, is today the most popular standard for mobile phone systems.

<sup>3</sup> Global Positioning System, GPS, is a satellite-based positioning system allowing you to determine your geographic position with an accuracy of some meters.

<sup>4</sup> Wireless Local Area Network, WLAN, is a standard for linking two or more computers using a wireless network device.

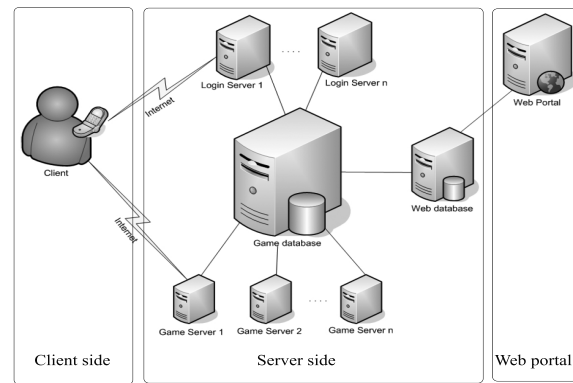


Figure 1: A system overview of the TGM project (Nilsson et al. 2007)

## 4 Research Design

### 4.1 Data Collection

Semi-structured interviews (Kvale 1997) were used to collect information on how students experienced the process of decision-making in the course.

An important requirement of the data collection method was that it should provide a rich data set where clues about the decision-making process could be found without exactly knowing in advance where to start looking for them. Semi-structured interviews are also well suited to a second-order research perspective.

Kvale (1997) describes semi-structured interviews as interviews where central themes and openings for relevant questions are prepared beforehand, but where it is also possible to adjust the order and formulation of the questions during the interview. The central themes and prepared questions can be seen as a desired structure, with the remainder of the interview comprising follow-up questions on interesting lines of thought from the initial answers.

An important goal with the phenomenographic research framework is to get the broadest possible set of experiences under the actual time constraints. The group for interview is selected not to capture all understandings, but to sample as broad as possible a range of experience in order to provide a rich data pool containing a wide range of experiences of the phenomenon. You cannot get all the understandings, since you can never see inside the minds of the group members. Students' backgrounds were surveyed in order to carefully choose the interviewees. Their academic records were examined to give a picture of their previous courses and achievements. The students were also asked to complete a questionnaire regarding their motives for participating in the project course, their expected achievements, and the personal skills they considered important. The gathered information was used to construct a profile of each student participating in the project course. Some of them turned up with similar academic background, personal skills, expectations, and motivations. Based on the assumption that diverse profiles were more likely to contribute to diverse experiences, 18 students were selected for interviews, four of whom were exchange students. This means that all but four students were selected for interviews, which certainly fulfilled the desire of a broad data set.

Decision-making processes might be different in the different project teams, therefore the students' experience is perceived differently. Since the phenomenon in focus, how the student experiences the process of decision-making, regards the full project

course, this difference is a part of the expected variation in experiences.

The interviews were performed in three sets of six interviews each over the duration of the course. The aim with this was to catch experiences from early, middle, and late team phases in the team development, as described thoroughly by Jaques (1995).

The interviews were held in either English or Swedish, according to the interviewees' preferences. The interviews were then processed and analyzed in their original language. Published excerpts will be presented in English and hence some translation is necessary.

The students investigated have different nationalities and genders. Although these factors might influence the empirical data, they have not been considered as a difference with regard to the analysis. Due to the integrity of the students, the fictitious names used in the excerpts might not suggest the same sex and nationality as the names in the original transcripts.

A final important note is that the study is not longitudinal. No comparisons were made between individual students or over time. This is consistent with the phenomenographic framework.

## 4.2 Analyzing Data

The full interviews were recorded on digital recorders. These methods for recording data during the interview are in accordance with how Kvale (1997) describes methods for collecting data from interviews. The recorded interviews were transcribed verbatim. An iterative process of identifying and categorizing the experiences followed. In this process, the researcher places statements from students in different categories, which are at first tentative. As the sorting process continues, the categories form their own contexts, giving a meaning to the different statements. The statements are continually re-sorted during this iterative process, since each newly added statement changes the meaning of the full set of categories. Finally, a set of categories is formed, each of which can be given a meaning in the researcher's own words.

Once this process was complete, the final categories were shown to a second researcher in order to establish their soundness. The categories and their meanings were also compared to the full interview transcripts in order to check their consistency.

## 4.3 Phenomenography and Learning

To reveal different ways of experiencing *how* students go about making decisions, a phenomenographic framework (Marton & Booth 1997) was used. Phenomenography is a research framework for revealing qualitatively different ways in which people experience, or learn about, a phenomenon.

A phenomenon, as the process of decision-making, can be experienced in many different ways. Marton and Booth describe phenomenography as a way to find and describe the outcome space that consists of the different ways of experiencing a particular phenomenon (Marton & Booth 1997). An important characteristic of a valid phenomenographic outcome space is the relationship between the categories. Berglund & Wiggberg (2006) describe this:

Since the categories illustrate different aspects of the *same* phenomenon, they are logically related to each other. Were they not, they would describe aspects of different phenomena. In general, some categories offer a wider or richer perspective and often come to embrace others in an inclusive structure.

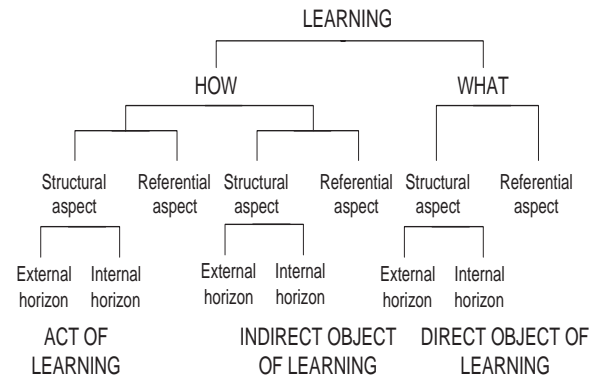


Figure 2: An analytical view of the experience of learning something Marton & Booth (1997, p. 91)

[...] the more embracing categories are generally more desirable. (Berglund & Wiggberg 2006, p. 266)

The complex process of learning is multi-faceted. In order to offer a framework for analyzing learning, phenomenography introduces a distinction between two aspects of learning:

(1) the *what* aspect of the learning, describing the content of the learning (for example a network protocol) and (2) the *how* aspect, describing how the students go about learning, or how they tackle their learning. While the first normally is referred to as the *object of learning*, the latter is labelled the *act of learning*. This distinction is, as Marton & Booth (1997) point out, purely analytical: the aspects can only be "thought apart" for research purposes and do not represent different concepts. (Berglund & Wiggberg 2006, p. 266)

Hence, even though the experience of learning something from the students' perspective is a whole process, phenomenography analytically helps us to analyze the process in different parts, the *what* and the *how*. The former deals with the content of learning, often referred to as the *direct object*, and the latter the *act of learning*.

The analytical distinction between *what* and *how* can be taken a step further by dividing the *how*-branch in two different parts, the *act of learning* and the *indirect object of learning*. The *act of learning* refers to how the students experience the learning. Berglund (2005) explains this *act of learning*:

The term "act" should here be interpreted in a broad sense, beyond the physical acts that a student performs in order to learn, such as reading a book, solving a problem and asking a friend. The term "act of learning" also includes abstract aspects, such as how students go about achieving their aims. (Berglund 2005, p. 42)

The *indirect object of learning* is about the quality of the act of learning, or what the act of learning aims at. This can also be seen of as the motive for learning (Berglund 2005).

While the above described terms form the main analytical separation in the experience of learning, the *act of learning*, *indirect object of learning*, and *direct object* can each in turn be divided in a *structural* and a *referential* aspect. The former denotes the structure by which we identify or recognize the

phenomenon, and the latter refers to the meaning of the experienced phenomenon. Again, this is just an analytical separation. The structure identified helps clarify the meaning, and the meaning helps us find the structure. A final analytical separation of the *structural* aspect helps to distinct between the phenomenon itself, its *internal horizon*, and its surroundings, its *external horizon*. Marton & Booth (1997) phrase this distinction like this:

That which surrounds the phenomenon experienced, including its contours, we call its external horizon. The parts and their relationships, together with the contours of the phenomenon, we call its internal horizon. (Marton & Booth 1997, p. 87)

See figure 2 for a summary of the analytical framework.

In this study the phenomenon is the process of decision-making, with a focus on *how* students experience that phenomenon. How students go about deciding things in the project is a strategy they adopt to be able to learn. This strategy can be seen as one of the “capabilities the learner is trying to master” (Marton & Booth 1997, 84) and thus the indirect object of learning.

A final remark on phenomenography is that it aims at gaining knowledge on variations in experiences on the collective level and not individual experiences. Marton & Booth (1997) put it like this:

[...] phenomenography focuses on variation. The objective of a study is to reveal the variation, captured in qualitatively distinct categories, of ways of experiencing the phenomenon in question, regardless of whether the differences are differences between individuals or within individuals. (Marton & Booth 1997, p. 124)

#### 4.3.1 Different Approach on Question B

Question B, addressing the kinds of decision structure that occur in the project, was a result of information that emerged during the data collection. Regarding question B, no particular analysis of the material has been performed, but the identified strategies have been categorized briefly. Answers to question B are presented in the empirical section as a collection of methods used by groups to make decisions.

### 5 Empirical Results

The empirical results consist of two sets of findings. For the question of how the student experiences the process of decision-making, a categorization is produced. Findings regarding the second question on structures for decision-making are also summarized. Together those set of findings describe the ways that students experience the process of decision-making as well as giving examples of the ways the group go about making decisions.

The first set of findings describes six categories that differ in their ‘richness’ or quality. The differences between categories include the size of the decision-making unit, the level of formalization of the decision-making process, and the level of involvement of people external to the group.

The categories are described in detail and illustrated by excerpts from the interviews. The presented excerpts are examples of the excerpts behind each category, and should give the reader an impression of the data supporting the category.

#### 5.1 Question on How Students Experiences Decision-making

The phenomenographic outcome space consists of six different categories describing different experiences of how the group handles decision-making. The categories all have different meanings (which phenomenographical terminology calls referential aspects), that give each of them a unique profile within the outcome space. Table 1 gives an overview of the referential aspect of each category.

We shall now describe each category in turn, giving examples of excerpts from each. Their focus, or structural aspect, and their meaning, or referential aspect, will also be described.

##### *Cat. 1: Decisions by Individuals*

In this category, individual decisions are expressed. That means that the decision either is too small or involves too few people to be handled by any means other than the individual first encountering the decision. The individual perceives the unimportance of the decision and therefore it becomes a private issue.

As an example constituting the base for this category, let us listen to Emma who states that most of the decisions are individual:

**Emma:** Most decisions have probably been made individually, [...], well there are lesser design decisions, maybe one, two or three persons have sat down in small groups and discussed how to design this or that thing, and it is these little decisions, small changes, [...], that in the end have created this project, then I think that many such decisions have been made individually, simply, that the largest absolute amount of decisions in the end have been individual.

In this excerpt, Emma goes on to describe different kinds of strategy for decision-making during the project, but indicates that the majority are individual.

Oscar continues by giving a reason for this when he tells us that those decisions often regard minor changes or minor things.

**Oscar:** Who’ll be affected, really, is it a decision that just concerns, affects one [...], if it is just a small function in what one is about to construct, then it is not necessary, maybe not to send it all the way up, it is not really anyone interested except the two that are implementing the detail.

It is worth noting that Oscar explains the informal way of making decisions where most of the issues are too small to bring up in whole group. Following this discussion, the focus in this category is therefore on one person and that particular individual’s decisions.

##### *Cat. 2: Decisions by Individuals with Preferential Right of Interpretation*

This category expresses an experience where a specific individual, namely the one who has responsibility for the result of something, also has the preferential right to decide. The decision-making therefore stays with that specific student. Edison gives us an explanation:

**Edison:** And for example for, I am, I am doing the communication with the client side and if the server goes wrong, and I am

	Label	(Referential aspect) Decision-making is understood as...	(Structural aspect)The focus is on...
1	Decisions by individuals	... too small or involving too few to be handled by any means other than the individual first encountering the decision. The individual perceives the unimportance of the decision and therefore it becomes a private issue.	...one person and that individual's decisions.
2	Decisions by individuals with preferential right of interpretation	...an informal right to interpret and make decisions even though the decision might involve other people.	...the knowledge that the decision may be of importance to other people in the project, but still it is recognized and treated as an individual decision.
3	Decisions by small group discussions	...a mutual agreement in a smaller group. The decision matters for more than just one individual and therefore automatically involves opinions from more people.	...on the smaller group and its discussion.
4	Decisions by group discussions supported by a facilitator	...group discussions supported by a facilitator.	...the facilitator and the small group.
5	Decisions by democracy in full team	...a democratic process in a formalized full team setting.	...the team as a formal body where strategies for structured decision-making are present.
6	Decisions by mutual agreement between team and external stakeholder	...democratic involving the full team, but in the same time the 'reality' is involved in some way.	...the full team and its context, namely the surrounding stakeholders and their interaction with the team.

Table 1: Categories describing the outcome space for how students experience the process of decision-making

in charge of every... everything, and I, of course I have the, the right to decide, eh, the architecture and stuff like that.

In this excerpt, Edison is clear on the link between responsibilities and decision-making.

Another note to make is that some kind of unspoken formal structure leads to the preferential right to decide, but that formal process is not agreed in advance. Courtney illustrates this in the interview:

**Courtney:** It is not formal, but it is like, eh, everybody, I do not know, eh, we, it is not written anywhere, but it is like we are working, because everyone, in charge of different things, you, of course, this is your job, and you, you, of course you should have the, eh, [decision/right to decide].

Another important characteristic of this category is the awareness that the decision might involve, or have an effect on, other people's work. The focus here is therefore the knowledge that the decision may be of importance to other people in the project, but still it is recognized and treated as an individual decision. This also implies a wider description of the decision-making process than in the previous category since the decision is now understood as something that will affect others. Even so, the individual makes the decision herself.

### ***Cat. 3: Decisions by Small Group Discussions***

This category contains experiences comprising small discussions at the workplace, often while people are still sitting at their computers. Pairs or small groups reason around specific issues while they work. The groups are limited in size and decisions, and the full

group is not a part of this category. The focus is therefore on the smaller group.

Let us hear how Eaton describes the core feature of this category, small group discussions:

**Eaton:** Yes, it depends on the way we work, very often we work in pairs, or perhaps in threes, and then we reason with each other to come up with a good solution and, eh, since we all sit in the same room.

The small group discussions are often centred on specific issues and seem to be task-oriented. Decision-making is therefore experienced as mutual agreement in a smaller group. The decision discussed is also something that matters for more than just one individual and therefore automatically involves opinions from more people. This makes the category wider than the previous one.

Ashley illustrates a situation where two different pairs of the project group had to solve something:

**Ashley:** And then when there has been things that are associated with both parts, or with the both parts in the project, then we may have had a meeting about this and then we've sat down and discussed it and thus have reached a joint decision.

As Ashley states, the decision affects more than just one person, and this is something that is acknowledged. The groups meet informally, though, and there are no traces of formal structures to choose between different options that arise from the discussions. Instead, the one arguing best wins.

### ***Cat. 4: Decisions by Group Discussions Supported by a Facilitator***

This category describes decision-making experiences where the project manager is involved, not as someone

who works with the particular issue in focus, but as a facilitator for the discussion. The group that gets facilitating support can be of any size. Ashley will help us again by describing such a situation to us:

**Ashley:** And she was also sort of part of the discussion, tossed ideas and such, since she's kind of well situated in everything.

**Interviewer:** Yes.

**Ashley:** But she said that, look, we have kind of discussed this for 15 minutes and, it was just a tiny detail. Because this was something that would take like between 5 and 20 minutes to implement. And then she sort of said that enough is enough.

The category involves situations where a group discusses a particular issue. The discussion need not be formalized or planned, but more than one person is involved.

Harold gives another example from this category:

**Harold:** [The project manager] has been there as a mediator if there hasn't been a solution [...] and then we've been forced to make a decision. And that is, has functioned well, I think.

The facilitator is here described as a driving force or arbitrator. The role is also emphasized as important for the progress of the project. The facilitator's involvement makes this category wider than the previous one, which involved only the small group.

#### ***Cat. 5: Decisions by Democracy in Full Team***

This category contains experiences of decision-making as a democratic process. The team has formalized a process in order to make decisions that people can recognize as fair. This category includes descriptions of formalized discussions where pros and cons are elaborated on. The focus is on the team as a formal body where strategies for structured decision-making are present.

Jake will start by telling us how he experiences the decision-making:

**Interviewer:** And the first thing I want to ask is how a decision is made in your team.

**Jake:** Yes, it is very democratic, eh, it is definitely not so that I decide everything, instead we discuss everything together. Eh, some minor decisions have been taken together with me [...] But that was just things that, eh, well, the time plan and such things and then it was not so that all wrote the project plan, but all big decisions about how we shall, eh, make the game and such things, all are part of it.

Examples from this category make clear references to democracy. The interviewees give us a picture of formalized whole team processes. Let us listen to Alfred who describes one example of this process:

**Interviewer:** And then, did you open up for a general discussion or...

**Alfred:** Oh, ok, yes, yes, everyone can speak for free, can, give their own opinion about the specific, the scope, and maybe we, eh, how you say, we, kind of vote, voted.

**Interviewer:** Voted.

**Alfred:** Yes, voted, kind of.

**Interviewer:** Ok. So, you voted finally, everyone had a possibility to say something.

**Alfred:** Yes.

**Interviewer:** And then you voted.

**Alfred:** Yes, that is, tradition.

**Interviewer:** Ok.

**Alfred:** In our team, everyone can say something.

In this illustration, the level of formalization is high and the team has adopted a system of voting to make the decision. In other cases, thorough analysis of the situation is the experienced strategy to let everyone be a part of the decision:

**Interviewer:** Right... When you say democratic, then you mean that...

**Isac:** That, eh, we, well, we sort of discuss it, we propose, eh... pros and cons sort of, okay, this should be the best, sort of. Just logic, like that. Not, yes but I'm best, I'm right. You are wrong.

To conclude this category, decision-making is understood as a formal and democratic decision-making process within the full team.

#### ***Cat. 6: Decisions by Mutual Agreement Between Team and External Stakeholder***

This final category describes experiences of decision-making where the decisions are not just the team's, but involve some external person. This means that the decision has to be taken by the team and agreed on by some external stakeholder. Decisions in this category are still democratic and involving the full team, but in addition, the 'reality' is involved in some way. This is a wider view on decision-making since it includes not only the team and its formal process but also an external person.

Leslie starts to illustrate this:

**Leslie:** But then we had to change it again recently, because we thought, or Patric [external stakeholder] thought that, eh, Nok.. some Nokia phones we had chosen may not be so good so we had to deselect them and choose something else, so now it's surely decided, but that, that, that is the type of issue that took a long time.

Thus, the external stakeholder plays an active role in the decision. Furthermore, the external stakeholder may disagree with the project team, regardless of where in the process the team is. Let us listen to an example of this.

**Isac:** [...] we sat and discussed for surely two hours yesterday.. And in the end we agreed on some things.. and then he [the external stakeholder] sent a mail later in the evening or this morning, and he said that he had changed his mind about things we had agreed about.

Another characteristic of this category is the experience that the external person has a strong mandate, not to mention the final say in a decision. Two excerpts from the interview with Jake illustrate this:

	Decision-making strategies
1	Small group meeting
2	Outside meeting
3	Full team meeting
4	Lottery
5	Voting

Table 2: Categories describing different strategies for decision-making

**Jake:** Because, eh, he is the customer and he wants to have things done his way and we want things our way, so within the group it has been fairly easy to make decisions together. We have been fairly unanimous, well I think we have, eh... but, yeah, it is just Chris [external stakeholder] that becomes like this, sort of compensating, then [...]

**Jake:** Because, or well it is in the end his product... although... but I don't agree with his decisions...

This veto on the external person's side is an important factor in the understanding of this category. Even though decision-making is performed with the full team and democratic processes, the external party has the final say and can therefore override the team.

Focus in this category is on the full team and its context, namely the surrounding stakeholders and their interaction with the team.

Despite the external interaction, the experiences in this category indicate that the whole team contributes to a formal decision-making process.

### 5.1.1 Structural Aspect

As mentioned earlier, the structural aspect holds the categories together and provides them with an order. This is an important characteristic of a valid phenomenographic outcome space (Berglund & Wigberg 2006, Marton & Booth 1997). The focus, or structural aspect, of the categories is on different aspects of the team or its members and context. The rightmost column in table 1 presents these different focuses.

As can be seen in table 1 the focus of the categories grows from the individual project member via the smaller and the full team to the team and its context. According to this categorization, the least sophisticated way to make decisions is to make them on your own. The most sophisticated is to include not only the full team but also its context, in this case the client. An important note here is that the least sophisticated strategy for decision-making can very well be adequate at times.

## 5.2 The Ways Student Team Works to Make Decisions

Regarding the question of what ways the student team works to make decisions, the empirical results form five categories. Each presented category describes a strategy to make decisions that can be observed by a person outside the team. The categories are simply describing different ways to make decisions that the students have revealed in the interviews, and no connections between the categories are claimed. In the presentation, excerpts from the interviews exemplify the categories. The different strategies are summarized in table 2.

### Cat. 1: Small Group Meeting

Depending on the effect on the project, subgroups of the team may handle the decision. Decisions regarding complex matters are sometimes best suited for small subgroups. Oscar explains:

**Oscar:** It also depends on, eh, there are different types of decisions, we have for instance split up in server and client groups, six in each, where there are two sub-project leaders, call it, and they have functioned exactly as I within their groups, like, eh, well, like project leaders in general for client and server respectively and how they've made decisions, it is up to them but they have simply often discussed in smaller groups of at most six persons, for decisions that now have been their part, when we will take big decisions that involve the whole project we can either be twelve persons discussing together [...]

### Cat. 2: Outside Meetings

In unusual cases, decisions are made outside the formal meetings. People with higher presence more often get the opportunity to attend these meetings. Roberta explains:

**Roberta:** It is like when everyone is, during lunch-time someone is drinking coke, someone is having their lunch and they just talk freely during this process they gain some decision.

**Interviewer:** So it is in the formal gathering of people?

**Roberta:** Yes. Also, not everyone, are, I mean, not everyone there, (?) talk with each other...

### Cat. 3: Full Team Meeting

Here planned meetings, formalized by the formal structure of a project manager, are present. Rules exist about the structure of the meeting and the project manager takes an active role in the meeting. Bob describes this:

**Bob:** Yeah, then it was more like a meeting, with the whole team, and then we sat and discussed tossing up a lot of ideas like that, and then it ended up with us making a decision concerning some of them.

The formal structure is emphasized by Roberta:

**Interviewer:** So tell me a little bit about those Monday meetings, eh, when you are discussing something, like a decision on design or something like that, how, how is the, how is the structure of the discussion. How is the decision made?

**Roberta:** Structure, as to the structure...

**Interviewer:** Is everyone giving his or her opinion and then the leader decides or is it in some other way.

**Roberta:** The leader always stands in front of that whiteboard.

**Interviewer:** Ok.

**Roberta:** And, he writes what we are going to discuss, the, the points, all the points and topics and he lists that on the whiteboard

and everyone discuss the topics one by one. Eh, some of the members they maybe not, I mean, they talk not, not, not that much, but most of the members they give their opinion.

#### *Cat. 4: Lottery*

This category consists of people's testimonials of lottery as a decision-making strategy. Lottery is often used in decisions regarding roles, i.e. when choosing between two persons. Donald gives an example of this strategy:

**Donald:** [...] and also happened some time before that one got a day to think about what one was interested in, but then during the decision process we were all gathered together and then we had the opportunity to say what we were interested in, eh, and if several wanted the same position there was a draw [...]

#### *Cat. 5: Voting*

Voting as decision-making strategy happens in full team meetings when making decisions that do not directly regard people. Alfred gives an example of this category (the excerpt is also used for illustration above):

**Interviewer:** And then, did you open up for a general discussion or...

**Alfred:** Oh, ok, yes, yes, everyone can speak for free, can, give their own opinion about the specific, the scope, and maybe we, eh, how you say, we, kind of vote, voted.

**Interviewer:** Voted.

**Alfred:** Yes, voted, kind of.

**Interviewer:** Ok. So, you voted finally, everyone had a possibility to say something.

**Alfred:** Yes.

**Interviewer:** And then you voted.

**Alfred:** Yes, that is, tradition.

**Interviewer:** Ok.

**Alfred:** In our team, everyone can say something.

## 6 Conclusion and Implications

The current study investigates how computer science students experience the process of decision-making in computer science projects. Six categories have been identified, describing how the students understand decision-making. The level of sophistication differs between the categories, where the first describes an experience of decision-making as individual decisions too small and unimportant to be handled by anyone other than the individual. At the other end is the experience of decision-making as a democratic process involving both the full team and the context that the team acts in. The other four categories are situated between these two.

The level of sophistication in the experience of decision-making does not necessarily connect to reasoning on what is better or worse. A programmer in the team has to make a lot of small decisions, and being forced to bring all those to the table would create an untenable situation and diminish the progress in the project. The other extreme, to let one single person decide everything, is not good either. Another implication is that one individual team member can

make decisions that affect not only the project, but also other people's work in it as described in category 2.

According to Barker et al. (1991), no single strategy for decision-making is thought to be best; instead the choice depends on team process factors. The current results presented therefore fit well with that result. The frustration in excessively static processes for team decision-making as reported by Barker et al. (1991) does not seem to apply to the teams we studied. Instead, they experienced a decision-making scheme that adapts to different situations.

One question worth looking into is how the design specification of the project affects the decision-making process. Does the design specification decide the implicit line between category 2 and 3? In addition, if so, are the students aware of that?

Situation awareness is an important component for decision-making (Wickens & Hollands 2000). The categories found show that in some cases, when people experience decision-making as an individual process, perhaps the situation awareness might be lower since only a single perspective is involved.

According to the categories that we have found, it seems that the nature of the decision determines how people experience decision-making. The number of different experiences, six, also shows a richness in how the team goes about its decision-making. A comparison of these conclusions with discussions presented by Shanteau (1992) on domains in decision-making is interesting. The categories found point at a decision-making process that would be identified as belonging to the good domain. The conclusion here, following Shanteau (1992), is that decision-making in the studied project is not necessarily hard and therefore might work well.

The full picture of the categories gives an interesting view. Some decision-making is done individually (category 1-2) and some is recognized as subject for team discussions (category 3-6). There is a fine line between these different strategies, and which way to go seems to be a decision for the individual project member.

The second result of this study, how the student team works in order to make decisions, shows diversity in decision-making strategies. This diversity is denoted as positive in Barker et al. (1991) since it helps the team to make decisions in different situations.

Decision-making is shown to be an active part of the computer science student project. Decision-making is an important part of running a project and thus it seems likely that what decisions are made, and how, will have substantial implications on the learning environment, and thus is a factor to consider. Exactly in what ways decision-making is related to the learning outcome is still an open question, but some important inferences can be drawn at this stage.

Different decision-making situations require different decision-making strategies and these end up in some of the six categories presented. As said, it cannot be considered better or worse to be in a certain category, but it affects the level of interaction in the process of decision-making. This means that the decision-making processes chosen affect the desired level of interaction among the students. Hence, decision-making processes likely determine the possible peer learning in the student project groups and therefore play an important role. Thus, learning what decisions are made and how the processes of decision-making are constructed is something that could contribute to peer learning and make it possible to configure more rewarding project settings.

By getting more knowledge on how decision-making processes occur, teachers can be aware of the

possible learning outcomes of their project course design. Decision-making will also be a parameter to consider when setting up the project courses. The different project methodologies and software development methods used in student project courses also play a role in how much and what decision-making will occur among the students.

Decision-making in computer science student projects may be influenced by the interpreted goal of the project. One opening for further work is an investigation of students' interpretation of goals with a project. A phenomenographic analysis of how students experience the goal of this computer science project course is currently under way.

## 7 Acknowledgements

Many thanks to UpCERG research group at the Department of Information Technology, Uppsala University, Sweden, and to the COMPSE group, Helsinki University of Technology, Helsinki, Finland.

## References

- Back, H., Engberg, R., Herdegard, J., Laaksonen, M., Lejon, R., Nibon, D., Sjöblom, V., Tano, G., Tian, J., Zhao, M., Zhanf, H. & Yuan, S. (2007), Developing a location based service for mobile phones, Technical report, Department of Information Technology, Uppsala University.
- Barker, L. J. (2005), When do group projects widen the student experience gap?, in 'ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 276–280.
- Barker, L. L., Wahlers, K. J., Watson, K. W. & Kibler, R. J. (1991), *Groups in process : an introduction to small group communication*, fourth edn, Prentice-Hall, Englewood Cliffs, New Jersey 07632.
- Berglund, A. (2005), *Learning computer systems in a distributed project course: The what, why, how and where*, Acta Universitatis Upsaliensis, Uppsala, Sweden.
- Berglund, A. & Wiggberg, M. (2006), Students learn CS in different ways: insights from an empirical study, in 'ITiCSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 265–269.
- Brown, J. & Dobbie, G. (1998), Software engineers aren't born in teams: Supporting team processes in software engineering project courses, in 'SEEP '98: Proceedings of the 1998 International Conference on Software Engineering: Education & Practice', IEEE Computer Society, Washington, DC, USA, p. 42.
- Coppit, D. & Haddox-Schatz, J. M. (2005), Large team projects in software engineering courses, in 'SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education', ACM Press, New York, NY, USA, pp. 137–141.
- Daniels, M. & Asplund, L. (2000), Multi-Level Project Work; A Study In Collaboration, in '30th ASEE/IEE Frontiers in Education Conferences', Kansas City, MO, USA, pp. F4C–11 – F4C–13.
- Danielsson, T., Olsson, M., Ohlsson, D., Wärme-gård, D., Wiggberg, M. & Carlström, J. (2006), A climbing robot for autonomous inspection of live power lines, in 'Proceedings of ASER06, 3rd International Workshop on Advances in Service Robotics', Vienna, Austria.
- Desanctis, G. & Gallupe, R. B. (1987), 'A foundation for the study of group decision support systems', *Manage. Sci.* **33**(5), 589–609.
- Einhorn, H. J. & Hogarth, R. M. (1978), 'Confidence in Judgment: Persistence of the Illusion of Validity', *Psychological Review* **85**(5), 395–416.
- Entwistle, N. (1977), 'Strategies of learning and studying: Recent research findings', *British Journal of Educational Studies* **25**(3), 225–238.
- Faulkner, X., Daniels, M. & Newman, I. (2006), Open ended group projects (OEGP): A way of including diversity in the IT curriculum, in G. Trajkovski, ed., 'Diversity in information technology education: Issues and controversies', Information Science Publishing, London, pp. 166–195.
- Fiest, G. J. (1997), *The Influence of Personality on Artistic and Scientific Creativity*, BPS Books, Leicester, UK, chapter Handbook of Creativity.
- Janis, I. L. (1989), *Crucial Decisions: Leadership in Policymaking and Crisis Management*, Free Press, New York.
- Jaques, D. (1995), *Learning in Groups*, 2 edn, Kogan Page Limited, London.
- Kvale, S. (1997), *Den kvalitativa forskningsintervjun*, Studentlitteratur, Lund.
- Leeper, R. (1989), Progressive project assignments in computer courses, in 'SIGCSE '89: Proceedings of the twentieth SIGCSE technical symposium on Computer science education', ACM Press, New York, NY, USA, pp. 88–92.
- Marton, F. & Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associate, Mahwah, NJ, USA.
- McShane, S. L. & Glinow, M. A. V. (2005), *Organizational behavior*, 3 edn, McGraw-Hill.
- Newman, I., Daniels, M. & Faulkner, X. (2003), Open ended group projects a 'tool' for more effective teaching, in 'ACE '03: Proceedings of the fifth Australasian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, pp. 95–103.
- Nilsson, C. (2006), Teazle Goes Mobile - Project Plan v1.5, Technical report, Department of Information Technology, Uppsala University.
- Nilsson, C., Dong, H., Chen, N., Söderlund, A., Pettersson, S., Lundmark, S., Hägglund, J., Andersson, A., Ramqvist, C. & Far, S. A. (2007), Teazle Goes Mobile, Technical report, Department of Information Technology, Uppsala University.
- Pettersson, P. (2006), 'Project course presentation', Online at <http://www.it.uu.se/edu/course/homepage/projektDV/ht06/02-gruppindelning.pdf>.
- Pettersson, P., Gällmo, O., Hessel, A. & Mokrushin, L. (2006), 'Project course webpage', Online at <http://www.it.uu.se/edu/course/homepage/projektDV/ht06/>.

- Raven, B. & French, J. (1960), *The bases of social power*, Harper and Row.
- Seat, E. & Lord, S. M. (1998), Enabling effective engineering teams: a program for teaching interaction skills, *in* 'Frontiers in Education Conference 1998. FIE '98. 28th Annual', Vol. 1, Tempe, AZ, pp. 246 – 251.
- Shanteau, J. (1992), 'Competence in Experts: The Role of Task Characteristics', *Organizational behavior and human decision poceesses* **53**(2), 252.
- Simon, H. A. (1955), 'A behavioral model of rational choice', *The Quarterly Journal of Economics* **69**(1), 99–118.
- Swets, J. & Pickett, R. M. (1982), *The evaluation of diagnostic systems*, Academic press, New York.
- The Joint Task Force for Computing Curricula (2005), 'Computing Curricula 2005', Online at [http://www.acm.org/education/curric\\_vols/CC2005-March06Final.pdf](http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf).
- Waite, W. M., Jackson, M. H., Diwan, A. & Leonardi, P. M. (2004), Student culture vs group work in computer science, *in* 'SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education', ACM Press, New York, NY, USA, pp. 12–16.
- Wickens, C. D. & Hollands, J. G. (2000), *Engineering Psychology and Human Performance*, Prentice Hall, Upper Saddle River, New Jersey 07458.
- Wiggberg, M. (2007), "I Think It's Better if Those Who Knows the Area Decides About It" - A Pilot Study Concerning Power in Student Project Groups in CS., *in* A. Berglund & M. Wiggberg, eds, 'Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling', Uppsala University, Uppsala, Sweden, pp. 132–135.

# SYSTEM PAPER



# VILLE – A Language-Independent Program Visualization Tool

**Teemu Rajala**

**Mikko-Jussi Laakso**

**Erkki Kaila**

**Tapio Salakoski**

Department of Information Technology

University of Turku

20014 Turku, Finland

{temira, milaak, ertaka, sala}@utu.fi

## Abstract

Visualization tools have proven to be useful for enhancing novice programmers' learning. However, existing tools are typically tied to particular programming languages, and tend to focus on low-level aspects of programming such as the changing values of variables during program code execution. In this paper we present a new program visualization tool, which provides a language-independent view of learning programming. Moreover, program execution can be viewed in two languages simultaneously. Complete with role information of variables, the tool supports the learning process at a more abstract level, thus emphasizing the similarities of basic programming concepts and syntax in all imperative programming languages.

**Keywords:** Language independency, teaching programming, novice programming, program visualization.

## 1 Introduction

Teaching programming has provided challenges for computer science education for many decades. Constructing and even understanding computer programs has proven to be highly non-trivial task for most learners (McCracken et al. 2001, Lister et al. 2004, Tenenberg et al. 2005). Many computer-based systems have been developed to aid the learning process, particularly for novice programmers. Existing systems use various visualizations and animation techniques to assist the learners in understanding the behaviour of program execution (Hundhausen et al. 2002).

In general, most visualization and animation systems are heavily dependent on a particular programming language, and can only visualize program execution in that language. However, the syntax and structure of basic programming concepts are very similar in all imperative programming languages. Those concepts include, for example, control structures (sequence, selection, and loops), statements, expressions, arrays, and methods. From a student's point of view it is not particularly important to learn how loops are defined and executed in

a particular programming language; it is far more important to understand the basic principles behind the loop structure regardless of the programming language in question.

Grandell et al. (2006) have argued that in programming courses for novices, the syntax of the programming language should be as simple as possible. Simple syntax allows students to focus on learning the very concept of programming instead of struggling with excessive syntax. Thus in our opinion a simple pseudo-language could be used effectively as a first teaching language. When using a pseudo-language, the algorithm as well as the corresponding program code can be represented on a higher level of abstraction, as Boada et al. (2004) and Stern et al. (1999) have stated. However, as Garner (2006) has noted, a pseudo-language is often perceived as a language that can't be interpreted or executed.

Another abstraction of learning programming is provided by the roles of variables. Sajaniemi (2002) has defined a taxonomy of roles of variables, based on their behaviour during the execution of programs. The concept can be utilized regardless of programming language or even programming paradigm. Sajaniemi and Kuittinen (2003) have noticed that using the role information of variables in basic programming courses improves the learning process of students by enhancing their understanding of the program.

VILLE is a language-independent program visualization tool providing an abstract view of programming. It can be used both in lectures and for independent learning. It has a built-in syntax editor with which users can add new languages to the tool or modify the syntax of built-in languages (currently including Java, C++, and a pseudo-language). The visualizations can be viewed in any of the defined languages. To emphasize the language independency, VILLE has a parallel view displaying a program in two languages simultaneously. It is possible to trace program execution line by line and monitor program outputs and changes in variable values. To make visualization more effective and easily interpretable, there is an automatically generated textual description of each code line, including the role information of variables. VILLE comes with a set of predefined examples, which can be easily extended. In addition, VILLE's predefined or user-defined examples can be published on the web, allowing students to engage with a learning session at any time and place.

The structure of this article is as follows: section 2 presents related work, previous studies and related systems. VILLE and its features are presented in section

3. Section 4 presents the discussion, and finally section 5 presents the conclusions in brief.

## 2 Related work

Defining visualization is not a simple task. As Petre (1995, p. 34) has noted: “the question is not ‘Is a picture worth a thousand words?’, but ‘Does a given picture convey the same thousand words to all viewers?’ ” Petre presents the concept of secondary cues, which provide additional information about visualizations. Ben-Ari (2001) claimed that graphical and textual descriptions have to be synchronized, because deciding which issues of the problem are relevant is a major problem for novice programmers. Naps et al. (2002) state that visualizations appear to be useful only if they can engage the learner into a learning session.

*Jeliot 3* (Figure 1) is a tool used in tracing the execution of Java programs. As the execution advances step by step, the evaluations of expressions are visualized with graphical symbols. *Jeliot 3* is designed mainly to support the learning process of novice programmers. Kannusmäki et al. (2004) evaluated *Jeliot 3* with qualitative methods and pointed out that only students without any previous programming skills were willing to use it. However, *Jeliot 3* improved the novices’ skills of perceiving if-statements and loops, understanding objects, and tracing errors from program code.

*JIVE* (Gestwicki & Jayaraman 2002) is a program visualization tool that in addition to code highlighting visualizes object structure and the calling sequence of methods. According to Gestwicki and Jayaraman, *JIVE* has proved to be a practical tool for program visualization and debugging.

*BlueJ* is an example of a *static* program visualization tool (Kölling et al. 2003). Unlike *dynamic* visualization tools such as *Jeliot 3*, *JIVE* and *VILLE*, static tools don’t visualize program execution step by step, but instead focus on visualizing program structure and the relations between program components. *BlueJ* has a class view showing relations between classes and an object dock containing all initialized objects. According to Kölling et al. (2003), *BlueJ* is well suited to teaching programming with an objects-first approach.

Over the past few decades, many visualization and animation based applications have been developed, including *JavaVis* (Oechsle & Schmitt 2002) which uses object and sequence diagrams as visualizations, *ALVIS LIVE!*, based on the WYSIWYC (What You See Is What You Code) model and direct manipulation of program structures (Hundhausen & Brown 2007), and *Raptor* (Carlisle et al. 2005), a programming environment that uses dataflow diagrams for visualization. *JHAVE* (Grissom et al. 2003), *BALSA-II* (Brown 1988), *ZEUS* (Brown 1991), *XTANGO* (Stasko 1992) and *TRAKLA2* (Malmi et al. 2004) are algorithm animation systems, focusing on visualizing data structures and algorithms. In recent studies (Grissom et al. 2003, Laakso et al. 2005a, Laakso et al. 2005b) algorithm animation systems have been successfully applied to teaching data structures and algorithms.

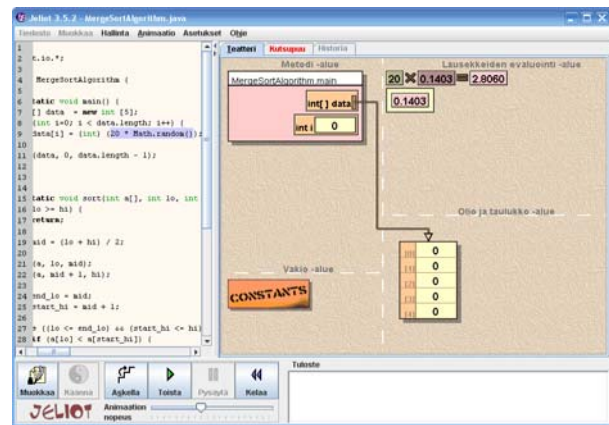


Figure 1: User interface of *Jeliot 3*

In conclusion, the tools most related to *VILLE* are *Jeliot 3* and *JIVE*, which have the same basic purpose and several common features. However, remarkable differences still exist in the abstraction level of visualization. The features of these three tools are compared in detail in section 4.

## 3 The VILLE tool

*VILLE* is a program visualization tool, which can be used to create and edit programming examples and to observe events in the examples during their execution. Its main purpose is to support the learning process of novice programmers. Teacher can add programming examples to *VILLE* and then visualize their execution in lectures or over the web.

### 3.1 Key features

In this section we present *VILLE*’s key features in four categories: level of abstraction, user interaction, tracing execution and customization. The categories reflect the main functions of features in this tool.

#### 3.1.1 Level of abstraction

**Language-independency.** One of the most important aspects of *VILLE* is the ability to view programming examples in several different programming languages. When observing program execution in different languages, a user can discover similarities in their basic functionalities. It is far more important for the novice programmer to learn how different programming concepts actually work than to focus on the syntactical issues of a specific language. We call this *the programming language independency paradigm*.

**Defining and adding new languages.** As built-in, *VILLE* supports Java, pseudocode and C++. The pseudocode’s definition can be altered to suit a teacher’s needs. It is also possible to define and add new programming languages to further extend the language support.

**The parallel view.** The program code execution can be viewed simultaneously in two different programming languages. This way the user can see how the execution

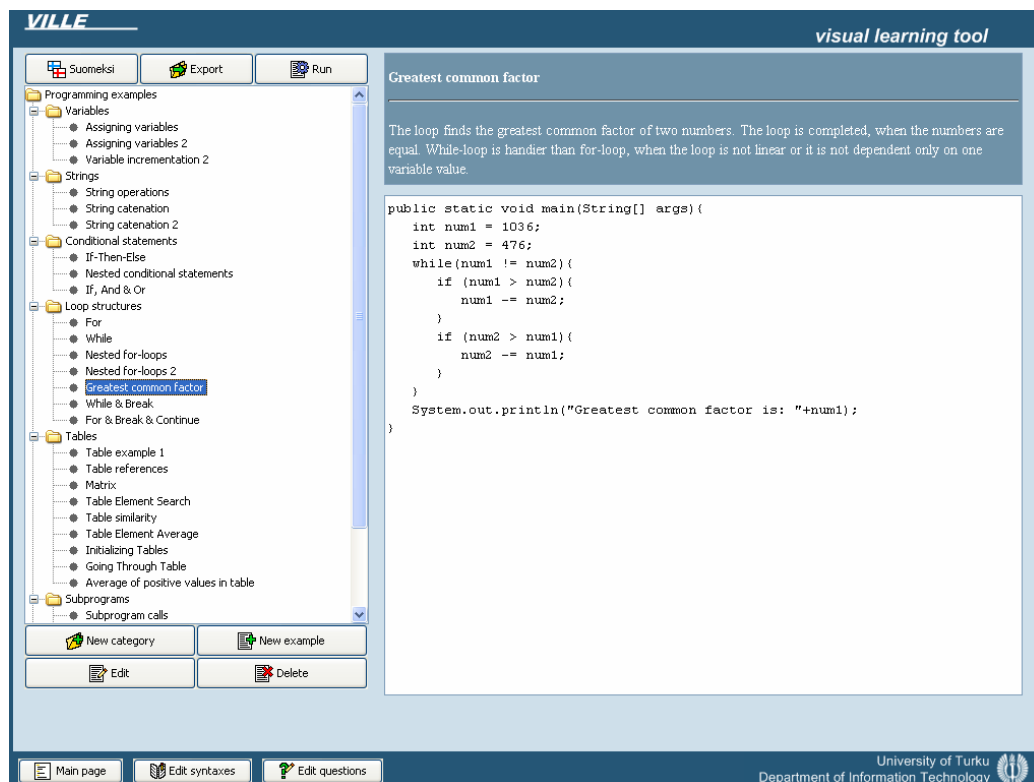


Figure 2: Main view of VILLE

progresses similarly regardless of syntactical differences between the languages.

**Role information.** The role information of variables is integrated into the code line explanation. According to Sajaniemi and Kuittinen (2003), the role information of variables helps learning and enhances understanding of the program.

### 3.1.2 User interaction

**Code editing.** Besides the example creation and editing view, the program code can also be edited in the visualization view, allowing users to trace the effects of changes in execution and visualization. The user's edits are not saved in the original example.

**Pop-up questions.** With the built-in editor the teacher can create multiple-choice questions and set them to trigger at certain states of the program execution.

**Flexible control of the visualization both forwards and backwards.** The user can move one step at a time, both forwards and backwards, in the execution of a program. Examples can also be run automatically with adjustable speed. Moving backwards in the program execution isn't usually possible in similar applications (e.g. Jeliot 3). Additionally, VILLE has an execution slider with which the user can progress to any state of the program execution.

### 3.1.3 Tracing execution

**Call stack.** The progress of the program execution between different methods due to function calls and returns is visualized with a call stack. When a method is called, a new window is opened on the call stack. The

window remains on the stack until the method is finished. When the execution returns to the caller, the return value is shown on top of the stack. The Call stack is especially useful in teaching *recursion*.

**Code line explanation.** Every code line has an automatically generated explanation, in which all the program events on the line are clearly explained. Furthermore, all possible outputs and variable states are shown. Code line explanation is a not a feature in most similar applications.

**Visualization row by row.** Progress of the program execution is visualized by highlighting rows in the code. In addition to highlighting the program row under execution, VILLE also highlights the previously executed row with a different colour. This makes the following of the program execution easier.

**Breakpoints.** The user can set breakpoints in program code lines and move between them, both forwards and backwards. This functionality enables debug-based control and observation of the program execution. Backward tracing between breakpoints is not a standard feature in program code debuggers.

### 3.1.4 Customization

**Example collection.** VILLE contains a predefined set of programming examples grouped into categories based on their subject. A user can create new categories and examples or edit the predefined ones. By creating and editing examples, the teacher can illustrate topics he thinks are essential in his programming courses.

**Publish examples.** With the export feature VILLE's examples can be saved to an example collection. The

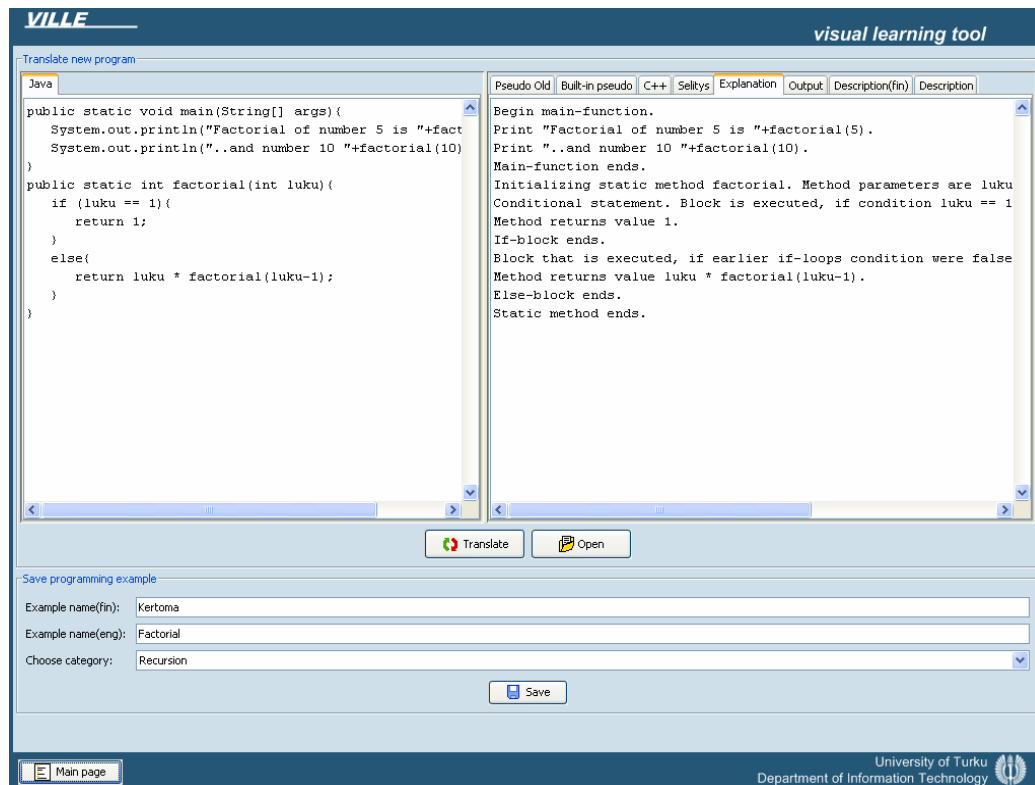


Figure 3: Creation and editing view of programming examples

example collection contains a version of VILLE with example creation and modification functions disabled; however, runtime modification is still enabled. The export feature can be used to publish a course's programming examples on the web for the students to use.

### 3.2 User interface of VILLE

VILLE's user interface consists of five separate views: the main view, the example creation and editing view, the visualization view, the syntax editor and the question editor.

#### 3.2.1 Main view

When the application starts, the main view is loaded. On the left side of the view (Figure 2) are the programming example tree and buttons for controlling the application. Users can modify examples with the buttons below. The buttons above the examples can be used to change the language of the application between Finnish and English, to export the examples to an example collection, and to move to the execution of the chosen example. The right side of the view displays the description and code listing of a chosen example.

#### 3.2.2 Example creation and editing view

In the creation and editing view (Figure 3) a user can add Java program code to the left text area; when the translate button is pressed, VILLE creates pseudocode and C++ translations (and, of course, translations to all the languages defined) and automatically generates explanations for each program line. The user can also write a general description for the programming example.

The translation of the program code is done with syntax definitions. There is a syntax definition for each programming language and also for the Finnish and English explanations. During the translation of the program, each code line is looked up from the Java syntax by using keywords, and then translated to other languages using the equivalent line in their syntax definitions. Thus, each language added to VILLE should define all the equivalent syntactical properties featured in VILLE's subset of Java syntax.

The events of a program code are solved by going through the program in its execution order and saving an execution event for each command. The events are used in the visualization view to control the visualization of the program execution.

The translation and execution tracing of programs is now possible only with Java. We are planning to add an option for translating code from the other defined languages in the near future. That will require a program component that parses the data stored in the variables of non-typed languages. After this the non-typed languages can be translated to Java, which can then be used in tracing the program execution events.

VILLE supports all the Java syntax necessary to teach introductory programming courses. It can handle the basic variable types, the main features of the String class, conditional statements, loop structures, tables and matrixes, methods, functions and records. With these programming concepts, the basic functionalities of programming can be well illustrated.

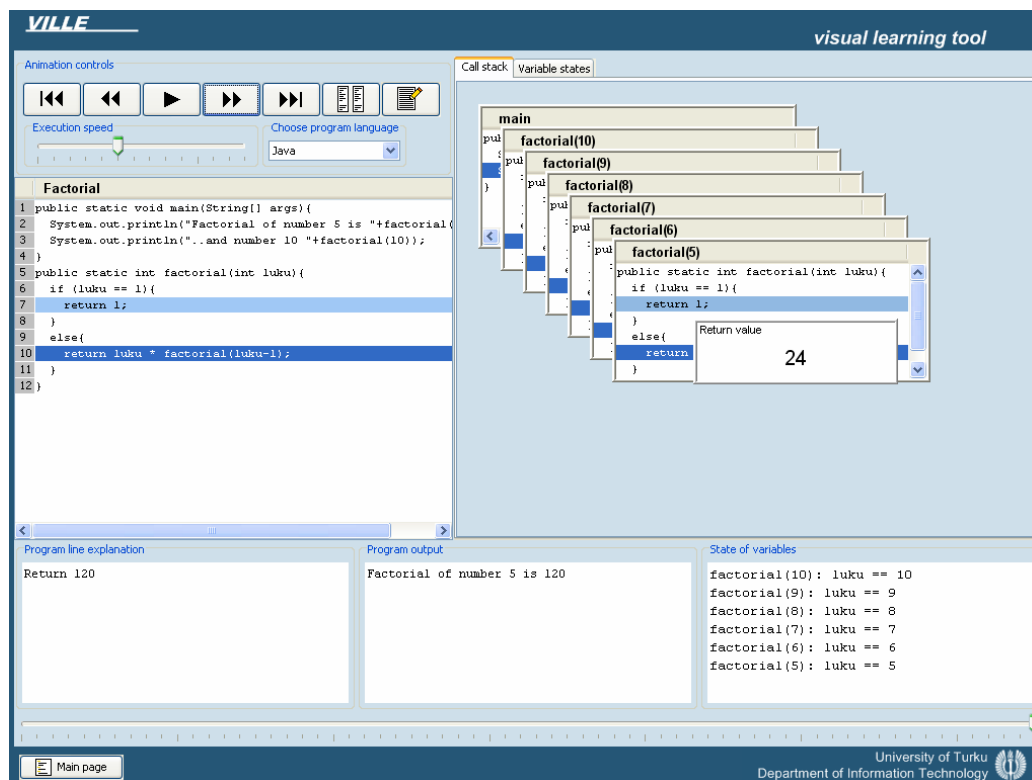


Figure 4: Visualization view of VILLE in call stack mode

### 3.2.3 Visualization view

In the visualization view (Figure 4) users can follow the execution of the programming examples. The control buttons for the visualization and the code listing of the programming example are located on the left side of the view. With the controls a user can start automatic program execution or alternatively move one step at a

time either forwards or backwards in the program. The user can also add breakpoints to any code line and move between the breakpoints with controls similar to debuggers.

The control area can also be used to change the program code language to any language defined, even during the execution. On the right side of the view lies the call stack,

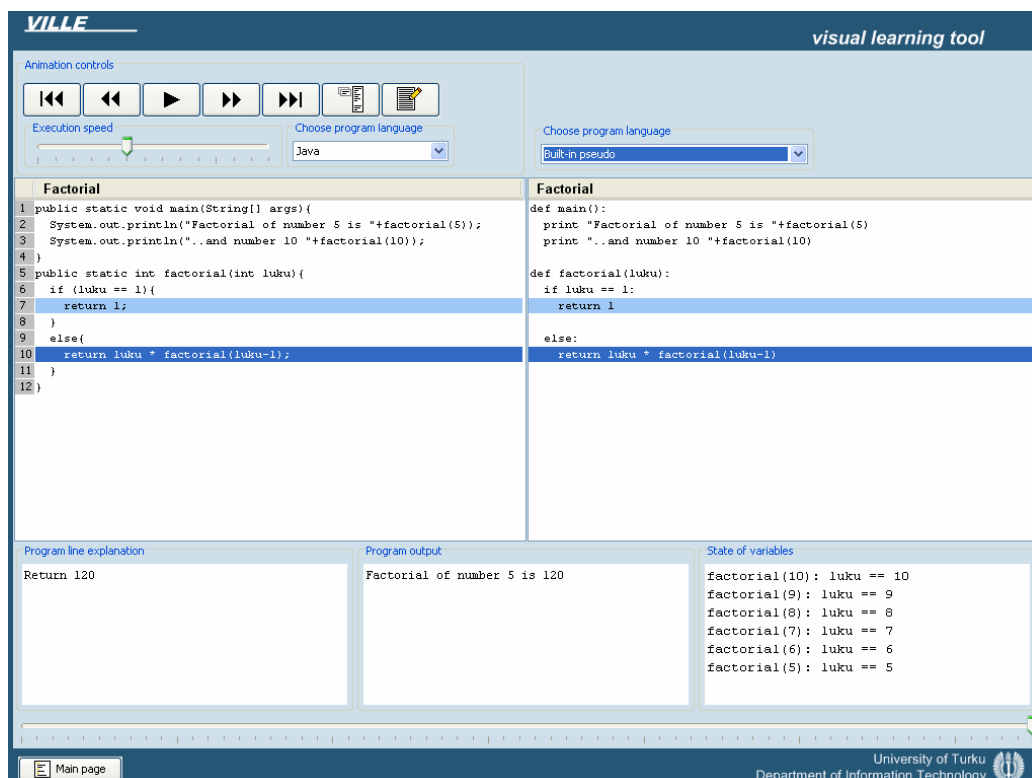


Figure 5: Visualization view of VILLE in parallel mode

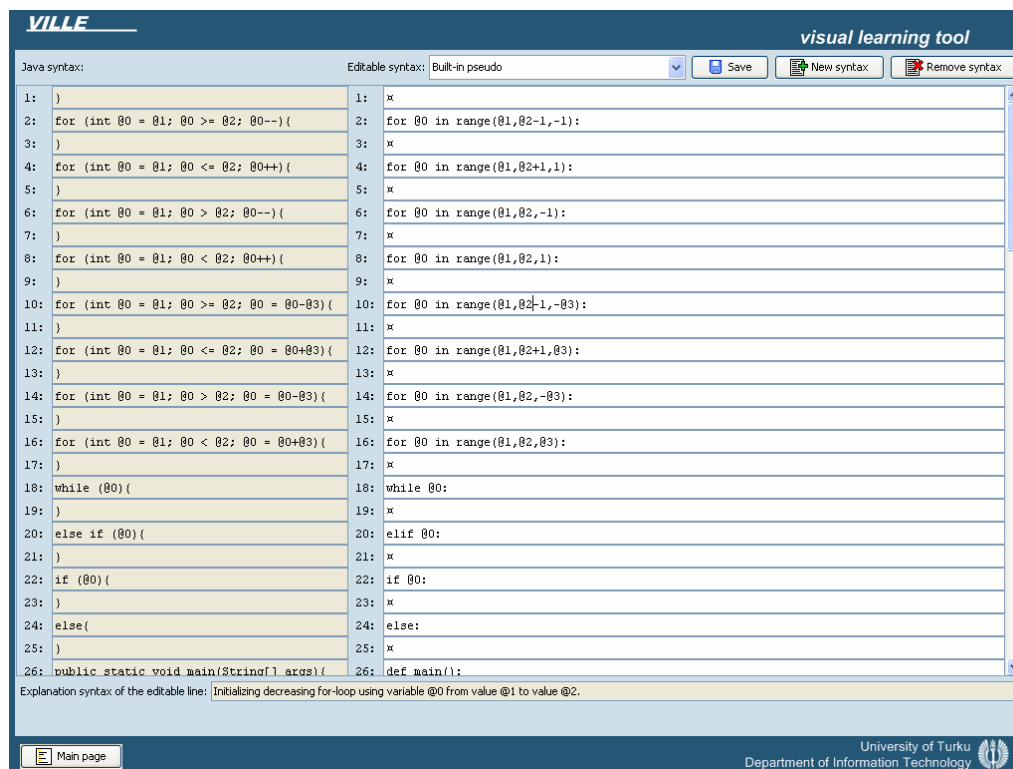


Figure 6: Syntax editor view of VILLE

on which the method calls are viewed in their own frames. The fields at the bottom of the view display the changes to program states, and the slider beneath those can be used to move around in the program execution.

The program execution in the visualization view can also be followed in so-called parallel mode (Figure 5), in which the program code is viewed in two selectable languages simultaneously; this way the syntax and the execution of the selected languages can be effectively compared.

### 3.2.4 Syntax editor

With the syntax editor (Figure 6) the teacher can add new programming languages to the system by defining their syntactical properties. The editor displays Java syntax lines on the left side. On the right side of the view, the user can select a syntax to modify or create completely new syntaxes. By comparing the Java syntax lines with matching lines in the modifiable syntax, the user can create new syntax lines understandable to VILLE.

### 3.2.5 Question editor

In the question editor view (Figure 7) a user can create multiple-choice questions and set them to trigger on selected code lines of a program. On the left side of the view the user can execute the program to a code line with controls similar to the visualization view, and then attach a multiple choice question to the code line. On the right side of the view the user can type in the question and the answer choices, select the choice count, and specify the right answer. All the created questions are displayed in the bottom right corner of the view.

## 4 Discussion

To enhance the learning process of novice programmers, the primary goal of VILLE has been to provide a higher level of abstraction by emphasizing the programming language independency paradigm. From the learner's point of view it is much more important to understand the principles behind basic programming concepts, such as loop control structures, regardless of the programming language. Thus, the use of a pseudo-language with less syntactical baggage than most actual programming languages is recommended for basic programming courses. With VILLE the teacher can define his own pseudo-language and then visualize program execution and its effects on the states of variables and the program output. However, because the program interpretation in VILLE is done with Java, the defined pseudo-languages should have corresponding program structures. The concept of the roles of variables gives a higher-level insight to programs, independent of programming paradigm, based on their variable behaviour. VILLE automatically generates a description with attached variable role information for every code line. This aids the interpretation of program execution as it acts as a secondary cue, aiding students in understanding the relations between programming concepts and program structures, which is essential in the process of learning to program.

Naps et al. (2002) have specified an engagement taxonomy that defines six different forms of learner engagement with visualization technology: 1) *no viewing*, 2) *viewing*, 3) *responding*, 4) *changing*, 5) *constructing* and 6) *presenting*. VILLE's feature set covers all these levels, except of course *no viewing*, which means that there is no visualization technology in use, and *changing*,

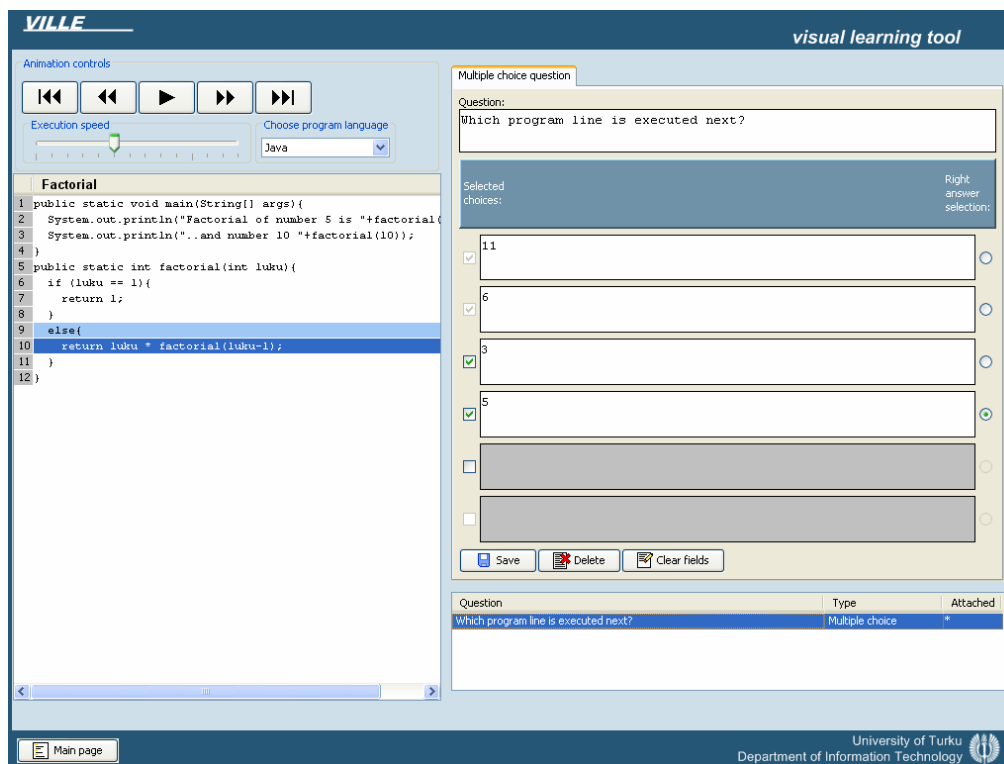


Figure 7: Question editor view of VILLE

which means that the system asks students for input to affect the execution of a program (this can, however, be achieved by altering the variable values in the program code). The majority of VILLE's features belong to the viewing category. Pop-up questions in the learner's perspective belong to the responding category. Students editing code in the visualization view and teachers creating new examples are clearly *constructing* visualizations, and one can engage in *presenting* just by demonstrating examples with VILLE to others.

To summarize, VILLE supports learning programming independent of the programming language. It offers customization features such as language and example creation, and provides interactivity by way of pop-up questions as well as interactive code editing to activate and engage the learner.

#### 4.1 VILLE vs. Jeliot 3

VILLE and Jeliot 3 are applications that can trace step by step a program code execution, but there are some differences between these two novel tools.

From the language perspective, Jeliot 3 supports only Java, while VILLE supports Java, C++ and a user-definable pseudo-language, and the language support is easily extensible. Moreover, a user can view the selected example simultaneously in parallel with two different programming languages and compare their syntaxes. This way we can emphasize the language independency paradigm which aids the process of changing from one programming language to other.

The controls are very similar in both applications, but VILLE makes it possible to step backwards in execution and to progress to any point of the execution directly with an execution slider. The absence of backward tracing is often frustrating when executing programs.

Jeliot 3 uses graphical symbols to visualize changes in variable states, and the execution of a single statement is presented with more detail than in VILLE, which presents variable states in a textual form. Both tools highlight the code line under execution, but VILLE also highlights the previously executed line to help the tracing of the execution.

VILLE automatically generates a description line for every executed program code line. The description includes the role information of variables and dynamic information about variable states. This helps students to interpret events in the executed code lines.

Jeliot 3 supports asking the users for input. This is not possible in VILLE. However, with VILLE, students can be asked questions during program execution, which is not possible in Jeliot 3.

VILLE includes predefined examples that can be used directly through the user interface. These examples can also be published on the internet as an example collection. This feature is not found in Jeliot 3.

Table 1 presents a comparison between VILLE and Jeliot 3. We have also included JIVE in the comparison, because it's quite similar to Jeliot 3.

	VILLE	Jeliot 3	JIVE
<b>General</b>			
Supported languages	Java, pseudocode and C++	Java	Java
Editable syntaxes	yes	no	no
Define new languages	yes	no	no
Examples	various built-in with a description; possible to add new ones	some included as files; possible to save new examples	possible to add new ones
Publish examples	yes	no	no
<b>Controls</b>			
Continuous running	yes	yes	yes
Adjustable speed	yes	yes	no
Reverse running	no	no	yes
Step forwards	yes	yes	yes
Step backwards	yes	no	yes
<b>Visualization</b>			
Call stack	yes	yes	yes
Program line explanation	yes	no	no
Graphical presentation of algorithm	no	no	yes
Variable values	yes	yes	yes
Role information of variables	yes	no	no
Program output	yes	yes	yes
Expression evaluation	verbal	graphical presentation	no
Program code viewed in	selectable language; alternative view with two languages	Java	Java
<b>Interaction with user</b>			
Editable programs in visualization state	yes	yes	no
'Stop-and-think' questions	with pop-ups	no	yes
Ask input from user	no	yes	no
<b>Technical implementation</b>			
Implementation language	Java	Java	Java
Compiles examples with	built-in compiler	DynamicJava	existing JVM
Data model	XML	ASCII file	Java bytecode

Table 1: comparison between VILLE, Jeliot 3 and JIVE

## 5 Conclusions

Learning to program is a challenging task, and a major step towards better learning is to go beyond syntactic features to understand the basic programming concepts. With this programming language independency paradigm, the similarities between the basic programming concepts in all imperative programming languages can be demonstrated, both syntactically and semantically. Furthermore, the understanding of the language independency principle should aid in adapting new programming languages and in changing from one language to another.

In the future, VILLE is going to be evaluated on the first programming courses at University of Turku. In addition to the learning performance the evaluation will focus on student engagement and the viability of VILLE's features.

In conclusion, VILLE promises an amendment to introductory programming courses by offering a chance to look at fundamental issues in an abstract way, and by allowing the teacher to create and use a programming language of his own.

## 6 Acknowledgment

This work was partially supported by Academy of Finland, project 121396, Automatic Assessment Technologies for Free Text and Programming Assignments.

## 7 References

- Ben-Ari, M. (2001). Program Visualization in Theory and Practice. *Informatik/Informatique* 2:8-11.
- Brown, M.H. (1988). Exploring Algorithms Using Balsa II. *IEEE Computer*, 21(5):14-36.
- Brown, M.H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. *In the Proceedings of IEEE Workshop on Visual Languages*, 4-9. New York: IEEE Computer Society Press.
- Boada I., Soler J., Prados F. and Poch J. (2004). A Teaching/Learning Support Tool for Introductory Programming Courses. *In the Proceedings of the Fifth International Conference on Information Technology*

- Based Higher Education and Training. ITHET 2004*, 604-609.
- Carlisle, M.C., Wilson, T.A., Humphries, J.W. and Hadfield, S.M. (2005). RAPTOR: A Visual Programming Environment for Teaching Algorithmic Problem Solving. *In the Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 176-180.
- Garner, S. (2006). The Development, Use and Evaluation of a Program Design Tool in the Learning and Teaching of Software Development. *Issues in Informing Science and Information Technology*, **3**:253-260.
- Gestwicki, P. and Jayaraman, B. (2002). Interactive visualization of Java programs. *In Proceedings of Symposia on Human Centric Computing Languages and Environments*, 226-235.
- Grandell, L., Peltomäki, M., Back, R.-J. and Salakoski, T. (2006). Why Complicate Things? Introducing Programming in High School Using Python. *In Proceedings of the 8th Australasian Conference on Computing Education*, Hobart, Australia, **52**:71-80.
- Grissom, S., McNally, M. and Naps, T. (2003). Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. *In Proceedings of the ACM Symposium on Software Visualization*, San Diego, California, s. 87-94.
- Hundhausen, C.D. and Brown, J.L. (2007). What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, **18**(1):22-47.
- Hundhausen, C.D., Douglas, S.A. and Stasko, J.D. (2002). A Meta-study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing* **13**:259-290.
- Kannusmäki, O., Moreno, A., Myller, N. and Sutinen, E. (2004). What a Novice Wants: Students Using Program Visualization in Distance Programming Course. *In Proceedings of the Third Program Visualization Workshop (PVW'04)*, Warwick, UK, 126-133.
- Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, **13**(4).
- Laakso, M.-J., Salakoski, T., Grandell, L., Qiu, X., Korhonen, A. and Malmi, L. (2005a). Multi-Perspective Study of Novice Learners Adopting the Visual Algorithm Simulation Exercise System TRAKLA2. *Informatics in Education*, **4**(1):49-68.
- Laakso, M.-J., Salakoski, T. and Korhonen, A. (2005b). The Feasibility of Automatic Assessment and Feedback. *In Proceedings of Cognition and Exploratory Learning in Digital Age (CELDA 2005)*. IEEE Technical Committee on Learning Technology and Japanese Society of Information and Systems in Education. Porto, Portugal, 113-122.
- Lister, R., Adams, S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin*, **36**(4):119-150.
- Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppälä, O. and Silvasti, P. (2004). Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2. *Informatics in Education*, **3**(2):267-288.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *ACM SIGCSE Bulletin*, **33**(4):125-140.
- Naps, T.L., Röbling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velázquez-Iturbide, J. Á. (2002). Exploring the Role of Visualization and Engagement in Computer Science Education. *In Working group reports from ITiCSE on Innovation and Technology in Computer Science Education*, **35**(2):131-152.
- Oechsle, R. and Schmitt, T. (2002). JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *In Diehl, S. (Ed.), Software Visualization. vol.2269 of Lecture Notes in Computer Science*. Springer-Verlag, 176-190.
- Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, **38**(6):33-44.
- 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*.
- Sajaniemi, J. and Kuittinen, M. (2003). Program Animation Based on the Roles of Variables. *In Proceedings of the 2003 ACM Symposium on Software Visualization*, San Diego, California, 7-ff.
- Stasko, J. (1992). Animating Algorithms with XTANGO. *ACM SIGACT News*, **23**(2):67-71.
- Stern, L., Søndergaard, H. and Naish, L. (1999). A Strategy for Managing Content Complexity in Algorithm Animation. *In Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, Cracow, Poland, 127-130.
- Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T.-Y., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R. and Monge, A. (2005). Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, **4**(1):143-162.



## DISCUSSION PAPERS



# Fighting the Student Dropout Rate with an Incremental Programming Assignment

Tuukka Ahoniemi

Essi Lahtinen

Teemu Erkkola

Institute of Software Systems  
Tampere University of Technology,  
PO Box 553, Tampere, Finland  
{tuukka.ahoniemi, essi.lahtinen, teemu.erkkola}@tut.fi

## Abstract

Large programming assignments can become huge obstacles to novice programmers, especially as teachers usually lack the resources to guide students sufficiently in-depth for the whole time. Changing the assignment to an incremental one consisting of smaller phases built on top of one another helps students to start in time, stay in time, and avoid succumbing to the huge workload.

We made the large assignment in a programming course incremental and got positive results when measuring the students' submission behaviour and their opinions on the phasing. The students felt that they were aided instead of just given more deadlines. The students willingly took advantage of our approach and really appreciated it. This article explains how we made our incremental assignment, how students used the phases, and how they felt about them.

**Keywords:** Novice programmers, programming education

## 1 Introduction

The bigger the programming assignments, the harder it is for students to grasp them. This is partly because novice programmers have difficulties in identifying the big picture and they approach the program line by line (Robins et al. 2003, Soloway & Spohrer 1989). The students may know the required programming concepts, but lack the skills to apply them (Lahtinen et al. 2005, Winslow 1996).

Novice students often also lack experience on the overall programming process, as they have only had experience in writing code fragments. They do not acknowledge the time required for proper testing and debugging, nor possibly even how to test or debug at all (Soloway & Spohrer 1989).

Teaching programming in general can be seen as an incremental process (Robins et al. 2003) — in a way it is natural to base new information on top of old. Some industrial programming schemes implement this idea in a systematic incremental approach.

In response to the known problems we decided to apply a similar incremental approach in a novice programming course that has a rather large programming assignment. We also wanted to study its effects on the dropout rate, the students' behaviour in applying the somewhat optional phases, and their perception of the incremental assignment and its workload. To

address these questions, we conducted surveys during the course.

In this paper we describe the problems encountered in our introductory programming course related to its relatively large programming assignment, and our solution of an incremental assignment. Then we introduce the ways we have measured the effectiveness of our experiment and present both the quantitative and qualitative results. The results are then discussed and conclusions are drawn.

## 2 The Course Setting

Our target course is the second introductory-level course in programming at Tampere University of Technology held in the spring semester for first-year students. After the course, which involves a few small programming assignments and one large one, students should know the very basics of object-oriented programming using C++.

### 2.1 The problems of the large programming assignment

The large programming assignment in our course tests the skills learnt through the whole course. It is large so that students can see the real benefit of using classes, get a grasp of the whole software process, and gain programming experience. The assignment should require intensive work over at least a couple of weeks, with careful design and testing also involved. The assignment is evaluated with automatic assessment, and when it is sufficiently correct, a teaching assistant manually gives it a summative grade.

Despite our explanations and urging, students have had a tendency to ignore the large size of the assignment. We have identified the following problems caused by this:

- Starting way too late
- A very hastily done implementation which merely *works*, but the student thinks it is perfect
- A very hastily done implementation which the student knows not to be adequate, but says he/she lacked the time to make it better
- A big dropout rate in the beginning (they don't know where to start) and in the last few days (they cannot get it working)
- Multiple desperate student requests for help in the last days before the deadline
- Many 'extratimers' (we have had a policy of giving few days' extra time with a grading penalty)
- Much negative feedback on the assignment size

We offer constant assistance throughout the course. As the students with most problems have usually started working too late, they do not take advantage of this support early enough.

To ease the start and guide the students to implement correctly designed solutions, the assignment has always begun with a *design phase*. Before they start programming the students have designed the classes/modules thoroughly, receiving feedback from a teaching assistant. After the design phase the students have had at least six weeks to implement the program. This has resulted in the problem that not all students have understood the need to be working bit by bit for the whole six weeks.

## 2.2 Incremental Assignment in Spring Semester 2007

To tackle the big problems we have faced with our assignment, we decided to apply the idea of building the program incrementally. The design phase was left as is, but the actual implementation part was divided into four incremental phases and a final submission that corresponded to the only submission of previous years.

The assignment description was released earlier, as a whole, describing what was to be done during the whole assignment. We also provided an additional phase submission specification which described what was to be implemented in each phase.

We designed the assignment to be easily divided into phases so that the next phase would require a working implementation of the earlier parts. The phases were still separate enough to be tested and debugged separately. As we wanted students to find the phases helpful rather than just adding further deadlines, we ruled the phases compulsory to the extent that skipping two phases in a row was not allowed (meaning that at least two phases were obligatory).

To encourage the students and to monitor their progress we gave them the opportunity to use automatic assessment for the phases. The requirements for these automated assessments were not as strict as those of the final submission, but missing a phase would give a student the extra responsibility of thoroughly validating the correctness of that phase. The different phases are explained in more depth in the following.

### 2.2.1 The phases

As in previous years, the task began with the design phase.

The first and second implementation phases held nothing actually new for the students. The first phase was only the start of the program, that is, the parsing and validating of the given program arguments. The second phase was to implement a completely functional command shell for the program so that it would understand which of the commands (and their arguments) are legal and which are not. We wanted the students to have a working command shell at this point to provide the means for testing the further functionality of the program.

The third implementation phase had the first tricky elements of the assignment. The main focus of this phase was on configuration file parsing. A completely correct file parser required precision and labour, and was supposed to construct objects of the used classes. The classes did not need to be fully implemented in this phase but the initialization and output functions were required to work correctly.

The fourth implementation phase was basically to complete the program. Though it may seem that this

would be the largest phase, this was actually the most convenient one. If the third phase worked (the classes were instantiated correctly) all that remained to students was the implementation of the rest of the methods — which they had already designed in the design phase.

The final submission had no more functionality than the fourth implementation phase. The automatic assessment tool was now testing each part of the complete program, which was also graded manually by a teaching assistant. The final submission also included also a documentation of the program. Only the design phase and the final submission contributed to the grade for the assignment.

## 3 Measuring the Effectiveness

To measure the effectiveness of the approach we collected data on all student submissions, presented a short survey after each phase, and at final submission time collected open feedback about the assignment as a whole.

When a student had passed the automatic assessment on a phase, the system asked the student the following multiple-choice questions:

- How easily did you manage through this phase?
- How clear is the implementation of the next phase for you right now?

## 4 Results

In this section we present results on the effects of our approach: quantitative results from submission statistics and student questionnaires and qualitative data from student feedback.

### 4.1 Submission statistics

The data collected from submissions give the total number of submissions, which is compared to the data from the previous year. For this year we have also derived ‘phase paths’ which show more precisely how students completed the phases. The average times of the submissions in each phase are also presented.

#### The dropout statistics

For measuring the student dropout rate during the large programming assignment (not during the whole course) we compared the number of students who attempted the design phase with the number who made final submissions. We also calculated the number of extratimers (students who submitted late). The results are shown in Table 1.

Table 1: Dropout statistics from 2006 and 2007

	2006	2007
Students in the design phase	240	196
Final submissions	183	143
of which extratimers	73	6
Percentage of dropouts	23.7%	27.0%
Percentage of extratimers	39.9%	4.2%

The results show that the plain dropout rate was not reduced at all, but actually increased slightly. However, the number of students who submitted late not only decreased but fell almost to nothing.

The ‘phase paths’ taken by students were analyzed in order to distinguish different student behaviours and how successful those behaviours were. Over 75%

of the students completed the first phase, which is already a great improvement, because it means that most of the students began implementing their program on time. Over 80% of these students passed the final submission. Only 47% of those who did not pass the first phase passed the final submission.

The second phase was seemingly as easy as the first one, as an almost equal number of students completed it. However, 83% of students who completed this phase, regardless of how they did in the first phase, went on to complete the entire assignment. This clearly supports our assumption that a phased exercise generally makes students start their work earlier, and that those who do so are much more likely to finish the work they have already started.

The third phase was without a doubt the hardest phase, as only 27% of all students submitted a work that passed it. Of these 27%, 64% passed every phase and the final submission.

The most popular paths through the phases were to complete every phase but the third (25.9% of the students) and to complete every phase (17.1% of the students). Surprisingly, the path that seemed to be of least work — completing only every other phase — attracted only 7.3% of students.

The more the students completed phases, the more likely they were to do the next phase. This is best seen in the case of the third phase. Roughly half the students who had completed both first and second phases also completed the third phase. Of those who had completed only the first phase, a little over one third completed the third phase. Of those students who had completed the second phase but not the first, only about 11% completed the third.

### Times of the submissions

In addition to the lower number of extratimers, students generally submitted their work earlier than usual. Table 2 shows that although the first phases were easier, students still clearly began working on them earlier than the last few days before deadline. Even the hardest phase — the third — was on average submitted more than 30 hours before the deadline. In comparison, the non-phased assignment of 2006 had a submission time average of 25 hours before deadline.

Table 2: Students' average submission times (hours before deadline)

Phase 1	Phase 2	Phase 3	Phase 4	Final
78	55	32	46	67

## 4.2 Student feedback

### Recognizing the hard part

As seen in Figure 1, after the first phase students generally knew what they were supposed to do next. Some of them still had not grasped the entire exercise description at this point and thus were not sure if they understood it correctly. After the second phase they realized that they had not done anything as big as the third phase before, and this is reflected in the greater uncertainty in the figure. However, after the third phase most of the students had a very clear idea of how to continue. These results are along the same lines as the results shown by the phase paths in terms of the difficulty of the phases, and reflect directly on the smaller number of submissions in the third phase.

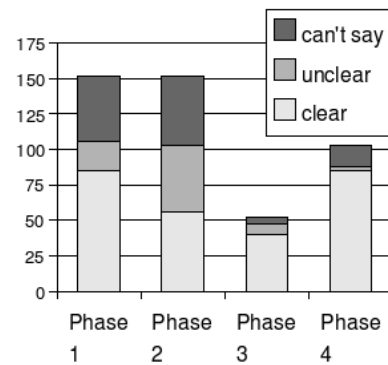


Figure 1: Students' answers to the question "How clear is the implementation of the next phase for you right now?" after each phase

### Effect on students' workload

Only 7% of the students thought that phasing increased the workload "a lot more" or "some more". In general the students didn't feel that their workload increased much — 21% thought that it increased a little — whereas 40% of the students experienced the phasing as guidance, hence lowering the amount of work required.

### Open feedback answers

We had 124 final surveys to analyze. Of these, 51 made no reference to the incremental assignment, leaving 73 that made some mention of the phasing. The results are shown in Table 3.

The open feedback results clearly show that the experience was really positive for the students. They felt that the phases were a huge aid for them in forcing them to start early enough with a good schedule and in helping to divide the assignment into reasonable parts. Most of the student complaints were about the uneven sizes of the phases — the third phase was surprisingly large. None of these students concluded that the incremental assignment was a bad thing.

## 5 Discussion

### Increasing the odds to success

An early start to the work seems to be a strong indicator for passing the assignment. Compared to our old system, 'starting early' means to start with phase 1 instead of waiting until phase 2. This seemed to be more attractive than to 'start working weeks earlier on a huge amount of work'. The students also started working early on the subsequent phases. One reason for this might be that the students have a clearer picture of what they are doing so it is easy to start well before each phase deadline. Anyway, some sort of change in attitude appears to have happened, since there were far fewer students who submitted in extra time.

Missing the first (or any) phase worked as a concrete message to students that their work was not on schedule. Because they had slacked off during the first phase, they now had to double-time to catch the others during the second phase. So although no actual penalty was given for missing a phase, the sheer need to work harder and faster may have done the job. We also emphasized in the lectures that these phases provide a schedule that we find realistic — in our opinion, missing a phase means you are late.

By phasing the exercise we were able to isolate the

Table 3: Results from the open feedback (n=73)

Overall evaluation on the incremental assignment mentioned really good	51
Overall evaluation on the incremental assignment mentioned fine / ok	4
Overall evaluation on the incremental assignment mentioned bad	0
Thanked for forcing to begin early and/or providing a ready schedule	42
...and in addition thought that would not have survived without the forcing	8
Thanked for providing ready-made division of the assignment	11
Did not like the ready-made division	1
Thanked for the ability to automatically test program in each phase	7
Complained about the uneven sizes of the phases	16
Complained about the schedule of the phases	8

harder part and let the students handle it separately. After the hard part the students knew how to finish the assignment. On the other hand, before the difficult phase they had already completed half of the assignment. Quitting at that point would be a pity.

One of the key aspects to a phased exercise is the way a student views it. As students tend to pick the path of least work, the success of phasing can be determined by how students take advantage of the system. In our case a very small percentage of students did only the minimum required submissions. As students themselves wrote in open feedback, phasing was for their help and they were pleased to take advantage of it.

Some of the students said that phasing actually increased their workload. This might be because, as many students confessed in the open feedback, they completed some of the phase submissions in such a rush that their result was not suitable as a basis for building anything new. Thus they had to refactor major parts of that phase for the next phase, resulting in excess work just because of additional deadlines. What these students did not state (and probably did not know) was that even without the additional deadlines they would probably have made many of the same mistakes, and would thus still have had to fix them before proceeding. Luckily, many of these students still recognized that they had at least learned valuable information when doing these refactorings.

### Guiding or restricting?

A single student pointed out that phasing spoiled the independent design of the program, forcing it into a single mould. However, the imposed structure did not differ greatly between the phased exercise of 2006-2007 and the non-phased exercise of 2005-2006. The actual assignment description was written in a similar way in both years. The phase description part of the 2007 assignment consisted mainly of a list of functionalities required for each phase — exactly the same things as in the assignment description — and some phase-specific output strings. The phases were also quite natural independent parts of the main assignment description. Therefore the only significant ‘forced’ parts of the phasing were starting early and writing the program in a smart, easily testable way. Students were given the option of making their own schedule, so this part of the assignment was in no way forced. The student might also have meant the elaborate guidance of the design phase. Good designs meant following the course teachings in regard to programming style and forbidden methods; to accept poor designs would have resulted in bad programming style, greater dropout rates because of bad design, and even more whining and complaints. Because of this, students with bad designs in the design phase were guided toward the intended solution for their own good.

## 6 Conclusions

Despite putting in a lot of effort designing and implementing an incremental programming assignment, we did not manage to lower the actual dropout rate during the assignment. However, we did see a dramatic reduction in the number of students submitting their work in extra time. The students started and finished early, and thus with better outcomes. Most students who dropped out did so during the first two phases. Thus there were fewer frustrated students who had worked for weeks without completing the assignment.

The students’ opinions on how the incremental assignment affected their workload were also encouraging. Only a minority perceived that the phases increased the workload, and many students thought that the guidance decreased their working hours. Overall the feedback about the incremental assignment was really positive and the students seemed to latch on to the phase deadlines instead of seeing them as obligatory extra work.

Building an incremental assignment requires additional work from the teacher. The phases should be designed carefully to be logical parts in a reasonable and logical schedule. An ‘easy start’ helps the students to start working early but the following phases should not be surprisingly larger than the prior ones, as our third phase was. The students should feel the phases are for their help — as indeed they are.

## 7 Acknowledgments

Nokia Foundation has funded part of this work.

## References

- Lahtinen, E., Ala-Mutka, K. & Järvinen, H.-M. (2005), ‘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* pp. 14–18.
- Robins, A., Rountree, J. & Rountree, N. (2003), ‘Learning and teaching programming: A review and discussion’, *Computer Science Education* **13**(2), 137–172.
- Soloway, E. & Spohrer, J. (1989), *Studying the Novice Programmer*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.
- Winslow, L. E. (1996), ‘Programming pedagogy – a psychological overview’, *SIGCSE Bulletin* **28**(3).

# Improving Mathematics and Programming Education – The IMPEd Initiative

**Ralph-Johan Back and Linda Mannila**

Dept. of IT, Åbo Akademi Univ.  
Turku Centre for Computer Science  
Joukahaisenkatu 3-5 A, Turku, Finland  
{backrj, linda.mannila}@abo.fi

**Mia Peltomäki and Tapio Salakoski**

Dept. of IT, University of Turku  
Turku Centre for Computer Science  
Joukahaisenkatu 3-5 B, Turku, Finland  
{mia.peltomaki, tapio.salakoski}@utu.fi

## Abstract

In this paper we discuss the topics that should be included in basic computing education and the levels of education at which they should be introduced. We present the resource centre *IMPED*, which aims at improving education in mathematics and programming by drawing on results from empirical studies. The research focuses on three main topics, which are also briefly summarized.

## 1 Introduction

Mathematical subjects in general and information technology in particular are vital for the success of the Finnish knowledge-based society. New technologies cannot be adopted without a sufficiently educated and trained workforce. This requires the entire school system to be taken into account, as high quality elementary and secondary school education are prerequisites for an excellent university system.

Both mathematics and computing rest essentially on a solid theoretical foundation. Whereas mathematics instruction at a comprehensive level understandably focuses on learning how to use basic mathematics in practice, high school mathematics is more theoretical. Mathematics is taught quite extensively at both comprehensive and high levels in the Finnish school system. For instance, a student choosing the advanced mathematics syllabus at high school level has 10 compulsory and three elective courses, and a student taking the general syllabus has six compulsory and two elective courses. In other words, all high school students have to take at least six courses in mathematics (Finnish National Board of Education 2003).

By contrast, the Finnish high school core curriculum does not include any courses on computing or any other similar topic. Instead, focus is put on learning how to use the computer and its applications as tools, and only a few high schools offer their students courses covering the more theoretical aspects of computing. Thus, although both mathematics and computing have comparable

theoretical foundations, this does not show in the way computing is introduced in education. On the other hand, the practical applicability of mathematics may also be overlooked if the courses are viewed as purely theoretical. The absence of this link between theory and practice in computing is also prevalent at university level (Almström et al. 2001).

In our opinion, combining mathematics and computing is a natural way of linking theory and practice together. It offers students an insight into the theoretical basis of computing and provides them with a more practical view of mathematics and one of its application areas.

Starting in 2000, researchers at the departments of IT at Åbo Akademi University and the University of Turku have worked closely together with the aim of developing a course sequence that could provide such a link. New methods have been developed and empirically evaluated in classroom settings.

As of fall this year (2007), the research activities are complemented by the resource centre *IMPED* (Improving Mathematics and Programming Education), aimed at disseminating our work to other educational institutions. The centre is a joint project between the IT departments at Åbo Akademi University and the University of Turku, led by Professor Ralph-Johan Back (Åbo Akademi University) and Professor Tapio Salakoski (University of Turku). The project is funded by the *Federation of Finnish Technology Industries*.

In this paper, we will briefly present the *IMPED* initiative and open up a discussion regarding the way that basic education in mathematics and IT could be organized at secondary and tertiary level to support the future of the Finnish knowledge-based society.

## 2 Teaching Mathematics and Programming

*IMPED* is supported by our active research which aims to improve the understanding of mathematics and programming among secondary school students and first-year students at universities and polytechnics. So far our research has focused on the following topics:

- Teaching mathematics with a precise notation and a logical foundation (structured derivations)
- Teaching programming using a simple language (Python)
- Teaching the construction of correct programs (invariant based programming)

In the following sections we will briefly describe each of these parts.

## 2.1 Mathematics with a Precise Notation

Exact formalism is usually avoided at lower levels of education such as high school mathematics courses. When not exposed to proofs and exact definitions, students are not given the opportunity to truly understand the very foundation of mathematics. Mathematical and logical reasoning remains a ‘secret’ and proofs become magic tricks when the formal parts are hidden. The reason for not covering formalism and proofs at lower levels is that these topics are considered too difficult and abstract. This might certainly be the case, if they are taught in the same way as they would be to experts or university students in their final years of study.

*Structured derivations* is a calculational proof format developed by Ralph-Johan Back and Joakim von Wright (Back et al. 1997; Back & von Wright 1999). They have extended the derivational-style proof approach as presented by Dijkstra (1990) and van Gasteren (1990) by adding nested derivations (subderivations), allowing inferences to be presented at different levels of detail. The method is thus theoretically well founded. A sample derivation is given in Figure 1.

These derivations introduce a new approach to teaching mathematics including exact formalisms and proofs. Structured derivations facilitate problem solving and enhance the possibilities of rereading and discussing solutions afterwards, as compared with traditional informal approaches to writing down solutions. The method is more rigorous and exact than the traditional methods used in secondary level mathematics, and it contributes not only to the preciseness of expression but also to more systematic and straightforward presentation

$$\begin{aligned}
 & \begin{cases} x + 2y = 4 \\ 2^x = 8^y \end{cases} \\
 \equiv & \text{ {simplify the second equation} } \\
 & \bullet \quad 2^x = 8^y \\
 \equiv & \text{ {rules of exponents} } \\
 & \quad 2^x = 2^{3y} \\
 \equiv & \text{ {the exponential function with base 2 is monotonic} } \\
 & \quad x = 3y \\
 \dots & \begin{cases} x + 2y = 4 \\ x = 3y \end{cases} \\
 \equiv & \text{ {assume the second equation and simplify the first one} } \\
 & \bullet \quad [x = 3y] \\
 & \quad x + 2y = 4 \\
 \equiv & \text{ {substitute from the second equation} } \\
 & \quad 3y + 2y = 4 \\
 \equiv & \text{ {equation solving} } \\
 & \quad y = \frac{4}{5} \\
 \dots & \begin{cases} y = \frac{4}{5} \\ x = 3y \end{cases} \\
 \equiv & \text{ {substitute } y \text{ from the first equation into the second one and calculate} } \\
 & \begin{cases} y = \frac{4}{5} \\ x = \frac{12}{5} \end{cases}
 \end{aligned}$$

Figure 1: Sample structured derivation

of mathematical reasoning.

The use of structured derivations in mathematics education has been extensively evaluated, starting in 2001, when a longitudinal study teaching the compulsory courses in advanced mathematics using structured derivations was initiated in a Finnish high school. The results have been encouraging, suggesting that this method has the potential to improve students’ performance in mathematics courses as well as their understanding of mathematical reasoning and problem solving (Back et al. 2004; Peltomäki & Salakoski 2004).

Starting in fall 2006, structured derivations are the standard approach used in the basic course on logic at the IT department at Åbo Akademi University. Currently, the evaluation of the use of structured derivations in education involves five institutions at both high school and university level.

## 2.2 Practical Programming

The question of which language to introduce to novices has been discussed quite extensively, and may ultimately be considered a matter of taste. When teaching programming, the focus should be on developing programming skills, not on the language or its syntax. However, in order to maximize the time spent on programming, one should avoid having to ‘waste’ time on additional and unnecessary language constructs.

Notational overhead and complex syntax render many of the popular languages unsuitable for novices. Java, for instance, is one of the most commonly used languages at universities today, but requires much extra code in order to complete even the simplest program. All time spent on such extras is time away from actual programming.

In 2004 we decided to teach a simpler language, and chose *Python*<sup>1</sup> for this purpose. Python is an interpreted high-level scripting language designed by Guido van Rossum as a general-purpose language but with education firmly in mind; van Rossum has even suggested that everybody could master programming using Python (van Rossum 1999). The language is freely available and has many of the features characteristic to a language suitable for teaching programming (Milbrandt 1993; Weinberg 1998), such as a small and clean syntax, an enforced structural design, dynamic typing and expressive semantics. The interpreter provides immediate feedback and there is an active user community providing books, tutorials, examples and other course material online.

Python comes with a large number of modules which provide extra functionality, and many of these can be used to make even small programs interesting and motivating. The code snippet in Figure 2 illustrates how the webbrowser module is used to let students write programs that open web pages in the default browser, while at the same time practicing important concepts

<sup>1</sup> See <http://www.python.org>.

```

import webbrowser

options = {"A" : "http://www.google.com",
          "B" : "http://www.yahoo.com",
          "C" : "http://www.altavista.com"}

for site in options:
    print site + " : " + options[site]

try:
    choice = raw_input("\nChoose a site: ")
    webbrowser.open(options[choice])
except:
    print "You did not pick a valid alternative."

```

**Figure 2: Sample Python program**

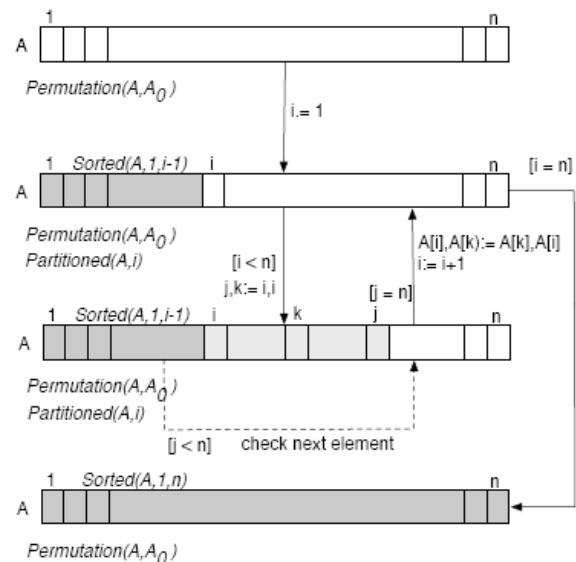
such as user input, iteration, dictionaries and exception handling.

We have now used Python in nearly 15 programming courses at secondary level, and the experiences have been positive (Grandell et al. 2006; Mannila et al. 2006). The benefits of teaching a simple language such as Python have also been pointed out by others (e.g. Agarwal & Agarwal 2006; Guzdial 2003; Miller & Ranum 2006; Radenski 2006). As a result of our experience from secondary level, the IT department at the University of Turku incorporated Python in one of its basic courses in 2006. Starting in fall 2007, the IT department at Åbo Akademi University made the switch from Java to Python as the language of instruction in the first programming course.

### 2.3 Constructing Correct Programs

Universities give various courses on logic, formal methods, program semantics etc. aiming at giving the students the skills needed to write correct programs. However, these are not directly linked to the practical programming courses in which students learn to write code. As a result, it is common that students do not apply the issues learnt in the theoretical courses when doing actual programming. Instead, students learning programming usually go about it by ‘trial and error’; they iteratively write code, test it, and modify it as needed. The modifications might introduce new errors into the code, calling in turn for further changes. This iterative process makes it difficult to create correct programs, since one can never prove that a program is correct by testing, one can only point out the errors one happens to find using some, perhaps arbitrarily chosen, test cases. Another approach is needed to learn to construct programs that are known to work.

*Invariant based programming* (Back 1983; Back 2005) is a diagrammatic hands-on approach to constructing correct programs. Similar ideas were earlier presented by Reynolds (1978) and van Emden (1979). In invariant based programming, the program invariants are formulated before the code is constructed. The process starts with drawing pictures illustrating the basic data structures involved and how they will be changed during execution of the algorithm (Figure 3). From the pictures, the initial, final and intermediate situations are identified. These situations are generalized and the program is constructed as a (nested) invariant diagram (Figure 4). The final program is correct if it is consistent (all



**Figure 3: Final version of the pictures drawn to illustrate the algorithm at work**

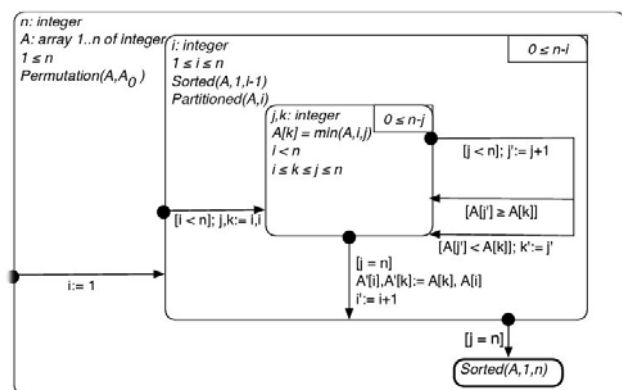
invariants are preserved), terminates (no infinite execution loops exist) and live (termination only occurs at final situations). The programs can be constructed and verified using only pen and paper, but tool support is also provided (SOCOS) (Back et al. 2006).

The invariant based approach was introduced in a first-year undergraduate course at Åbo Akademi University in spring 2007 with positive results (Back 2007; Back et al. 2007).

### 3 Putting the Pieces Together

The approaches to teaching mathematics and programming presented in the previous section work well together and give a solid foundation in both the theory and practice of programming. For instance, a course covering structured derivations and another covering practical programming in Python give students all the background knowledge they need to successfully complete a course covering invariant based programming.

Although the approaches are well suited for creating a continuum of courses, they can also be introduced separately. Structured derivations can be used solely as a



**Figure 4: Final version of the invariant diagram for selection sort**

way for improving students' understanding of mathematics, and invariant based programming can be introduced to students who have background knowledge in a proof format other than structured derivations and a programming language other than Python.

#### 4 Discussion

In our opinion, there is a need for a closer link between theory and practice in CS education, and in this paper we have briefly presented the IMPED resource centre and the research and activities that take place within the centre aimed at bridging the gap between theory and practice. To what degree should the problem be addressed at university level and to what degree at lower levels? What background knowledge would be desirable from students enrolling for CS studies? And should we, as educators in the computing field, take actions in order for computer science to be reintroduced as an independent subject at high school level? In that case, what measures would be appropriate?

#### 5 References

- Agarwal, K. K. and Agarwal, A. (2006), Simply Python for CS0, 'J. Comput. Small Coll.', vol. 21, no. 4, pp. 162-170.
- Almstrum, V. L., Dean, C. N., Goelman, D., Hilburn, T. B., and Smith, J. (2001), Support for teaching formal methods, SIGCSE Bulletin vol. 33, no. 2, pp. 71-88.
- Back, R.-J. (1983), Invariant based programs and their correctness, in W. Biermann, G. Guiho & Y. Kodrato, eds, 'Automatic Program Construction Techniques', MacMillan Publishing Company, pp. 223 – 242.
- Back, R.-J. (2005), Invariant based programming revisited, Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland.
- Back, R.-J. (2007), Basic Approach and Teaching Experiences. To appear in 'Formal Aspects of Computing Science'.
- Back, R.-J., Eriksson, J. & Mannila, L. (2007), Teaching the Construction of Correct Programs Using Invariant Based Programming. Accepted to the '3<sup>rd</sup> South-East European Workshop on Formal Methods', Thessaloniki, Nov. 30 – Dec. 1, 2007.
- Back, R.-J., Eriksson, J. & Myreen, M. (2006), Verifying invariant based programs in the SOCOS environment, in 'Teaching Formal Methods: Practice and Experience'. BCS-FACS.
- Back, R.-J., Grundy, J. & von Wright, J. (1997), 'Structured calculation proof', Formal Aspects of Computing, 9, pp. 469 – 483.
- Back R.-J., Peltomäki, M. & Salakoski T. (2004), Structured Derivations Supporting High-School Mathematics, in 'Current research on mathematics and science education', Univ. of Helsinki, pp. 104 – 122.
- Back, R.-J. & von Wright, J. (1999), A method for teaching rigorous mathematical reasoning, in 'Proceedings of Int. Conference on Technology of Mathematics', University of Plymouth, UK.
- Dijkstra, E. W. & Scholten, C. S. (1990), Predicate Calculus and Program Semantics, 'Texts and Monographs in Computer Science', pp. 21 – 29.
- Finnish National Board of Education (2003), Lukion opetusuunnitelman perusteet (Principles of the Upper Secondary School Curriculum, in Finnish), Valtion painatuskeskus, Helsinki.
- Grandell, L., Peltomäki, M., Back, R., & Salakoski, T. (2006), Why complicate things?: introducing programming in high school using Python. In 'Proceedings of the 8th Australian Conference on Computing Education', pp. 71 – 80.
- Guzdial, M. (2003), A media computation course for non-majors. In 'Proceedings of the 8th Annual Conference on innovation and Technology in Computer Science Education' (Thessaloniki, Greece, June 30 - July 02, 2003), ACM, New York, NY, pp. 104-108.
- Mannila, L., Peltomäki, M. & Salakoski, T. (2006), What About a Simple Language? Analyzing the Difficulties in Learning to Program. 'Computer Science Education', vol. 16, no. 3, 2006, pp. 211 – 228.
- Milbrandt, G. (1993), Using problem solving to teach a programming language in computer studies, 'Journal of Computer Science Education', 8(2), pp. 14 – 19.
- Miller, B. and Ranum, D. (2006), Freedom to succeed: a three course introductory sequence using Python and Java, 'J. Comput. Small Coll.', vol. 22, no. 1, pp. 106 – 116.
- Peltomäki M. & Salakoski T. (2004), Strict Logical Notation Is Not a Part of the Problem but a Part of the Solution for Teaching High-School Mathematics, in 'Proceedings of Kolin Kolistelut – Koli Calling. The 4<sup>th</sup> Annual Finnish/Baltic Sea Conference on Computer Science Education', pp. 116 – 120.
- Radenski, A. (2006), "Python first": a lab-based digital introduction to computer science. In 'Proceedings of the 11th ITiCSE ' (Bologna, Italy, June 26 - 28, 2006), ACM, New York, NY, pp. 197 – 201.
- Reynolds, J. C. (1978), Programming with transition diagrams, in D. Gries, ed., 'Programming Methodology', Springer Verlag, Berlin, pp. 159 – 165.
- van Emden, M. H. (1979), Programming with verification conditions, 'IEEE Transactions on Software Engineering', volume SE-5, number 2, pp. 148 – 159.
- van Gasteren, A. J. (1990), On the Shape of Mathematical Arguments, 'Lecture Notes in Computer Science', pp. 90 – 120.
- van Rossum, G. (1999), Computer Programming for Everybody, Corporation for National Research Initiatives, CNRI Proposal #90120-1a. Available online: <http://www.python.org/doc/essays/cp4e.html>.
- Weinberg, G. M. (1998), 'The Psychology of Computer Programming', Dorset House Publishing Company.

# Debating the OO debate: Where is the problem?

**Anders Berglund**

Uppsala Computing Education Research Group  
Department of Information Technology  
P.O. Box 337  
SE- 751 05 Uppsala  
Sweden

Anders.Berglund@it.uu.se

**Raymond Lister**

Faculty of Information Technology  
University of Technology Sydney  
P.O. Box 123,  
Broadway 2007, Sydney  
Australia

raymond@it.uts.edu.au

## Abstract

In this paper we discuss problems related to the teaching of object-oriented programming (OOP). We argue that more research on how the computer science teacher understands OOP would be beneficial. Our argument takes its point of departure in three sets of studies: (1) an ongoing study on how computer science teachers understand core concepts of OOP, (2) a study of how the teaching of OOP is discussed within the CS community, and (3) a set of studies that discuss the different ways in which CS teachers experience their teaching. This paper reports on an ongoing study of the different ways in which computing science teachers understand object-oriented programming, and what they mean when use the term *objects first*. The phenomenographic research approach has been applied to the analysis of a discussion that occurred in the SIGCSE-members mailing list. Two understandings of *objects first* have been identified: (1) as an extension of imperative programming, and (2) as conceptually different from imperative programming. These two understandings are illustrated via the differing ways in which computing science teachers use the term *polymorphism*.

**Keywords:** Object-oriented programming, objects-first, phenomenography.

## 1 Introduction

Object-oriented programming (OOP) is hard for students to learn and hard for teachers to teach. The learning and teaching of OOP is often discussed at Computing Education Research (CER) conferences, and articles about it appear regularly in the CER journals.

Many attempts have been made to improve the learning and teaching of OOP. Among these initiatives, to name but a few, are: environments for teaching Java (Kölling, Quig, Patterson, & Rosenberg, 2003); research into the students' experience of learning OOP (Bruce et al., 2004; Eckerdal & Berglund, 2005); explorations through the lenses of various learning theories (Ben-Ari, 2001, 2004;

Robins, Rountree, & Rountree, 2003); analysis of the properties of languages (Mannila, Peltomäki, & Salakoski, 2006); and changes to teaching approaches (Pedroni & Meyer, 2006). Despite these many initiatives, the learning and teaching of OOP remains a problem, with low passing rates and high attrition rates.

## 2 Broadening the perspective on the debate

In this paper we argue that our teaching community would benefit by broadening the current debate, which emphasises the technology of OOP and the learning of CS students, to include an examination of ourselves, the teachers, and our own understanding of what OOP means. We will base our argument on an ongoing project on the various ways in which CS teachers understand some core concepts in OOP, as illustrated in an analysis of a 2004 SIGCSE mailing list debate on objects early (Lister et al., 2006).

### 2.1 How do CS teachers understand core concepts?

In the third and fourth weeks of March 2004, there was a vigorous discussion about teaching OOP on the SIGCSE-members mailing list (SIGCSE, 2004a & 2004b). A list of the 99 postings is given elsewhere (Lister et al., 2006b). The discussion focused on when object orientation should be introduced in beginners' programming courses: should objects be introduced early (*objects first*), or should objects be preceded by imperative programming (*imperative first*)? An ITiCSE working group, chaired by the two authors of this paper and Tony Clear, Auckland University of Technology, New Zealand, studied this discussion from several different research perspectives (Lister et al., 2006).

When reading and re-reading the mailing list postings, we came to realize that the discussants understood fundamental concepts in different ways and thus put different meanings into terms such as *objects first* and *polymorphism*. To explore this question, we decided to do a phenomenographic analysis of *how the list discussants understood objects first*.

#### 2.1.1 Phenomenography

Phenomenography is an empirically based, pedagogically anchored research approach, aimed at exploring how something is understood by a group of people (Marton & Booth, 1997). The outcome of a phenomenographic

research project is an ordered description of the different meanings that the phenomenon under investigation (in our study *objects first*) has for the members in the group. Phenomenography is a qualitative, non-positivistic approach, which during the past few years has come to play an important role in CER (Berglund, 2006; Berglund, Box, Eckerdal, Lister, & Pears, 2008).

### 2.1.2 *Objects first* is understood in two ways by the discussants

Our preliminary results show that *objects first* is understood in two different ways, corresponding to two phenomenographic categories. We wish to stress that these categories do not express the main theme of the debate, objects first or imperative first. They ‘only’ describe different meanings of the first of these stands.

*Objects first* can be understood

1. as an extension of imperative programming,
2. as something conceptually different from imperative programming.

### 2.1.3 Understanding the categories

The phenomenographic categories are constructs that summarise and ‘abstract’ different meanings. They do not illustrate individuals; an individual can see something in one or many ways. A category can be analysed into (or ‘is constituted by’) different Dimensions of Variation (DoV) or parameters, where each DoV can take certain ‘atomic’ values, or be un-instantiated.

Table 1 shows the two categories, their dimensions of variation and the values of these dimensions. Since the purpose of this paper is to encourage a debate, rather than to present the final outcome of a research project, we will here only illustrate the values of one DoV, *Polymorphism*, with two quotes, one corresponding to each value of the DoV, and thereby to different categories. A further discussion on the empirical data will be published in the future.

The first, by McConnell, illustrates *Polymorphism understood as different objects*:

I still think that selection, repetition, variables, arrays, are still critical foundational CS1 topics. [...] I firmly believe that students leaving a CS1 class should have these foundational topics first but with an understanding of objects. Decker

and Hirshfield's "The Object Concept" had its problems, but I think it is a good book because: (1) its of a reasonable size, (2) it introduces objects early, (3) it concentrates on foundational topics, and (4) it introduces "advanced" object concepts, such as operator overloading, inheritance, and polymorphism at the end.

We interpret the second, by Joe Berger, as an indication of the understanding *Objects interact polymorphically*:

Note that this is consistent with my thesis that polymorphism "means" that an object just knows what it is and behaves like it does without fuss or bother. [...] Note that I don't explain all this to students unless they ask (rarely) in any early course. Polymorphism just "works" consistent with the "object is in control" metaphor ...

As predicted by phenomenography, there is a hierarchical relationship between the categories. The second category is more advanced (in the phenomenographic sense) than the first, since the second presupposes the first – if imperative programming is not known to someone, then it is impossible for that person to view OOP as something different from imperative programming,. However, no one (at least today) needs to understand *objects first* in order to understand imperative programming. The phenomenographic theory predicts this kind of hierarchical structure. It indicates that the categories are related and thus that they are categories of the same phenomenon.

### 2.1.4 Our interpretation of the findings

We argue that these different ways of understanding *objects first* not only relate to teaching, but also, and more importantly, describe different ways of understanding object-orientation. We base this argument in the content of the categories, in our reading of the mailing list discussion and in the hierarchical structure of the categories. The arguments for *objects first* that are given along the lines of category one are not open to an interpretation of object-orientation as an interaction or a calculation on its own right. Thus they show a more delimited view of object-orientation.

Although these results are preliminary, we do not expect any important changes in our future work. We will investigate further concepts and might ‘fine-tune’ the description of the categories and elaborate on their constituents. We thus believe that the fact that *objects*

		Category 1. Objects first as an extension of imperative programming	Category 2. Objects first as something conceptually different from imperative programming
DoV1	Program execution	Objects are passive and are used when the program is run	Objects are active. Object interaction gives the algorithm
DoV2	Polymorphism	Polymorphism as different objects	Objects interact polymorphically
DoV3	Modifications	Modification as changing code	Modifications as adding to holes and hooks

Table 1. The dimensions of variation of *objects first*. In the column of each category, the values of the Dimensions of Variation (DoV) are shown.

*first* has two fundamentally different meanings, and the fact that there are corresponding underlying meanings in the interpretation of object-orientation, are stable.

The question we now ask ourselves is in what ways these contradictions of the understanding of object-oriented programming within our community influence our teaching and, ultimately, our students.

One could argue that our different interpretations do not constitute a problem for our students' learning, since they normally meet only one teacher and thus only one point of view. Our answer to this imaginary argument is related to the integrity of computer science. If it does not matter which of these interpretations our students meet, what is then the core of computer science?

## 2.2 Insights from an analysis of the objects early debate

In the phenomenographic (Marton & Booth, 1997) portion<sup>1</sup> of the previously mentioned multiple research perspective analysis of the e-mail discussion (Lister et al., 2006), we explored what the different arguments in the debate focused upon. The purpose was to reveal what the discussants 'talked about', and through this to better understand both the debate and its topic.

According to our analysis, the arguments in favour of an early introduction focused on three major, incommensurable, *themes*: (a) a narrow domain in focus, (b) a broad domain in focus, and (c) pedagogy in focus. Each of these themes could then be argued for, or understood, in four qualitatively different ways. For example, arguments in theme *a* could be categorised into the following categories: (a1) particular Java features, (a2) specific CS constructions, (a3) teaching, and (a4) students as students. In the paper describing these findings, we also demonstrate that the categories form a hierarchical structure and that the discussions in favour of imperative early can also be described according to a similar pattern.

In our conclusions, we argue that this structure helps in understanding the debate:

With this structure of themes and categories, it is easy to see that several misunderstandings have their origins in a bad match between two arguments. That is, two participants might believe they disagree, when in fact they are arguing about different things. For example, one participant might be focusing on language features, while another is arguing about an aspect of pedagogy.

The twelve categories, and the interrelationships of the categories, reveal the complexity of this discussion. It is not a discussion solely about programming languages or about how OO should be taught. Statements arguing that the 'solution' lies in a single concept (such as for

example a new teaching tool) are oversimplifications. The reality is more complex. A broad range of questions need to be further analysed and discussed before a community consensus will emerge. (Lister et al., 2006, p. 156)

## 2.3 Teachers' experience of their teaching

In a meta-study, Kember (1997) reviews and condenses a set of independent studies on how teachers experience their own teaching. He states that this body of research shows a distinction between two broad orientations<sup>2</sup>: teacher-centred/content-oriented and student-centred/learning-oriented. Recently these orientations have been confirmed for teachers in CS (Lister et al., 2007; Pears et al., in press).

Kember argues, from a phenomenographic perspective, that the student-centred approach is more advanced, or more complex, in that it presupposes the teacher-centred approach. To focus on the student a teacher must be capable of taking a step 'outside' herself<sup>3</sup> and seeing her acts not as an aim in itself, but in relation to the student. The rather few studies that have quantified these orientations with individual teachers confirm that the student-focused orientation is less common than the teacher-focused one.

The insights from Kember's work tell us that the attitude of the teacher is important for how she teaches. It is worth exploring what it is that makes some teachers take the step to see their teaching and the object of their teaching from the perspective of their students. By learning about this development, we learn something important about the CS teacher.

## 3 Summary

We believe that our discussions in section 2 pose more questions than they answer about our community and our own relation to OOP. The common thread in these three discussions is the community. In section 2.1 we demonstrate that the members of the community, the teachers, understand key concepts in different ways. In the following section (2.2), our argument is that the debate that takes place between the members of the community is often carried out in a way that is too 'simple' to capture the complexity of the issue. Finally, in section 2.3, we show that we know surprisingly little about the teacher, despite far-reaching evidence of her importance.

<sup>2</sup> Kember uses the term *orientation*. An alternative term, developed from the language of computer science and object-oriented programming, is *super-category*.

<sup>3</sup> We have chosen to refer to a teacher as "her" throughout this paper. Certainly, our claims are as valid (or as invalid) for a male teacher.

<sup>1</sup> This portion of the paper was mainly authored by Anders Berglund, with the support of Raymond Lister.

## 4 Open questions

Our insight, when working on the different constituents that form this paper, is that more research is needed concerning the CS teacher, her understanding of herself, her thoughts about her students, and the relationship between these entities. More precisely, what is it that we need to learn about the teacher? And how should such research be performed?

## 5 References

- Ben-Ari, M. (2001). Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45 - 73.
- Ben-Ari, M. (2004). Situated Learning in Computer Science Education. *Computer Science Education*, 14(2), 85 - 100.
- Berglund, A. (2006). Phenomenography as a way to research learning in computing. *Bulletin of the National Advisory Committee on Computing Qualifications, BACIT*, 4(1).
- Berglund, A., Box, I., Eckerdal, A., Lister, R., & Pears, A. (2008). *Learning educational research methods through collaborative research: The PhICER initiative*. In Simon & M. Hamilton (Eds.), Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008), Wollongong, NSW, Australia. CRPIT, 78, 35 - 42.
- Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodly, I. (2004). Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, 3, 143 - 160.
- Eckerdal, A., & Berglund, A. (2005). *What Does It Take to Learn 'Programming Thinking'?* In Proceedings of the 1st International Computing Education Research (ICER) Workshop, Seattle, WA, USA, 135 -143.
- Kember, D. (1997). A reconceptualisation of the research into university academics' conceptions of teaching. *Learning and instruction*, 7(3), 255 - 275.
- Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ System and its Pedagogy *Computer Science Education*, 13(4), 240 - 268.
- Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., Bower, M., Carbone, A., Davey, B., de Raadt, M., Doyle, B., Fitzgerald, S., Mannila, L., Kutay, C., et al. (2007). Differing Ways that Computing Academics Understand Teaching. *Australian Computer Science Communications*, 29(5), 97-106.
- Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., & Whalley, J. (2006). Research Perspectives on the Objects-Early Debate. *SIGCSE Bulletin Inroads*, 38(4), 173 - 192.
- Lister et al. (2006b) ITiCSE 2006 Working Group: Research Perspectives on the Objects-Early Debate. <http://wwwstaff.it.uts.edu.au/~raymond/iticse06workinggroup/> [Feb. 2008]
- Mannila, L., Peltomäki, M., & Salakoski, T. (2006). What about a simple language? Analyzing the difficulties in learning to program *Computer Science Education*, 16(3), 211 - 227.
- Marton, F., & Booth, S. (1997). *Learning and awareness*. Mahwah, New Jersey, USA: Lawrence Erlbaum Associates.
- Pears, A., Berglund, A., Eckerdal, A., East, P., Kinnunen, P., Malmi, L., McCartney, R., Moström, J. E., Murphy, L., Ratcliffe, M., Schulte, C., Simon, B., Stamouli, I., & Thomas, L. (in press). *What's the Problem? Teacher's experience of student learning*. In Proceedings of the 7th Baltic Sea Conference on Computing Education Research, Koli Calling, Koli, Joensuu, Finland.
- Pedroni, M., & Meyer, B. (2006). *The inverted curriculum in practice*. In Proceedings of the 37th SIGCSE technical symposium on Computer science education Houston, TX, USA, 481 - 485
- Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education* 13(2), 137 - 172.
- SIGCSE (2004a) SIGCSE-MEMBERS Archives March 2004, Week 3. <http://listserv.acm.org/scripts/wa.exe?A1=ind0403c&L=sigcse-members> [February 2008]
- SIGCSE (2004b) SIGCSE-MEMBERS Archives March 2004, Week 4. <http://listserv.acm.org/scripts/wa.exe?A1=ind0403d&L=sigcse-members> [February 2008]

# A doctoral course in research methods in computing education research

## How should we teach it?

**Anders Berglund<sup>1</sup>**

Uppsala University  
Uppsala Computing Education Research Group, UpCERG  
Department of Information Technology  
Uppsala, Sweden

[Anders.Berglund@it.uu.se](mailto:Anders.Berglund@it.uu.se)

**Päivi Kinnunen**

Helsinki University of Technology  
COMPSER  
Department of Computer Science and Engineering  
Espoo, Finland

[pakinnun@cs.hut.fi](mailto:pakinnun@cs.hut.fi), [lma@cs.hut.fi](mailto:lma@cs.hut.fi)

**Lauri Malmi**

### Abstract

This discussion paper considers how research methods could be taught in a doctoral computing education research course. We consider the challenges to course planning entailed by the nature of CER as a field, as well as the diversity of the students' backgrounds and its consequences for the course. We describe the research methods course that was given in spring 2007 at Helsinki University of the Technology, and some lessons learned from that course. Finally, we set some questions to stimulate a discussion.

**Keywords:** Computer Science Education Research, Computing Education Research, research methods.

### 1 Introduction

The objective of this paper is to stimulate discussion on research methods education in CER: what we should teach and how we should teach it to students. This paper does not present a course with detailed description of exercises but highlights some more general types of challenge that the teacher and students in CER may face on a research methods course. The three authors of this paper taught the course *Methods and results in computing education research* at Helsinki University of Technology (HUT), Espoo, Finland, during spring 2007. The course was not only a learning occasion for the 23 participants, but it also made us, as teachers and organizers, ask ourselves questions about the foundations of computer science education research and how it should be taught.

Computing Education Research (CER) is by its very nature a cross-disciplinary field, encompassing a diverse set of disciplines such as pedagogy, sociology and

philosophy together with its core in computer science. This diversity is richness, since the different disciplines offer inspiration as well as theories and methodologies of various characters. However, while it is enriching for the researcher, it becomes a challenge when designing doctoral education in the field, since each of the contributing disciplines has its own language, values, norms, and traditions – its own culture.

CER cannot yet meet this influence from other disciplines by referring to its own tradition and culture; it has simply not yet reached the kind of maturity that one finds, for example, within core computer science. Currently, it applies methods and uses theories from other sciences to a large degree, but work is carried out to support building the research field and community. This includes identification of the most important problem areas, such as has been done in the work by Fincher and Petre (2004). They provide an overview of the whole field, splitting it into 10 subfields. Their work was continued by Pears, Seidman, Eney, Kinnunen, & Malmi (2005) who aimed at identifying some core literature for the field. Valentine (2004) takes his point of departure in cognitive research and distinguishes between research activities and developmental, teaching-based activities in computing education. The categorisation proposed by Berglund (2005) is based on research approaches and methods, along with the theoretical frameworks that are used in a field.

Several activities aiming to form the CER community can be listed. Specific conference series dedicated to the field, such as the International Computing Education Research Workshop<sup>2</sup> (ICER) and the Baltic Sea Conference on Computing Education Research – Koli Calling<sup>3</sup>, have been launched in recent years. More specific workshops, such as Phenomenography in Computing Education Research<sup>4</sup> (PhiCER), have been arranged to support the enriching of research methods in CER (Berglund & Lister, 2008). Such activities, among others, support building the research community by mutual engagement,

---

<sup>1</sup> Anders Berglund is also currently affiliated to Helsinki University of Technology, COMPSEER, Department of Computer Science and Engineering.

Copyright © 2008, Australian Computer Society, Inc. This paper appeared at 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. Raymond Lister and Simon, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

---

<sup>2</sup> <http://www.cc.gatech.edu/conferences/icer2007/>

<sup>3</sup> <http://cs.joensuu.fi/kolistelut/>

<sup>4</sup> <http://www.it.uu.se/research/group/upcerg/PhiCER>

joint enterprises and a shared repertoire, as formulated by Fincher & Tenenberg (2006), building on Wenger (1998).

The course we have been teaching can be understood as an effort in this direction. We aim at organizing research training, especially on research methods and their applicability to different problems in CER. We see such research training as an essential part of community building, since the new doctoral students, coming from varying backgrounds, need to get an understanding of the complexity and richness of the whole field.

In this paper, we share our considerations of the nature of our field, as well as our experience of organizing a doctoral course in CER. Several questions are raised which we hope will be discussed within the community.

## 2 Challenges in Teaching Computing Education Research

The aim of Computing Education Research is often described in pragmatic terms. However, this is by no means the only possible focus. For example, the first lines of the call for papers for the third ICER workshop indicate a more theoretical stance:

Computing education, as a research discipline, is the study of how people come to understand computational processes and devices, and how to improve that understanding.

As we discussed in the introduction, several attempts have been made to define the content of CER. We state that the ways in which CER is often described, as a catalogue of its constituents instead of being discussed as a whole, shows that the criteria formulated by Fincher & Tenenberg (2006) have not yet been met.

Certainly, it is a challenge to design doctoral education in a field with these characteristics. Firstly, CER has no research tradition of its own on which courses can be built. Computer Science has a strong tradition, but its methods normally apply only to the more technically oriented research subfields of CER, such as development of tools and learning environment technologies. Secondly, researchers entering the field have varying backgrounds. Most of them come from some computing discipline but there are people with backgrounds in education, sociology, etc. Their previous research training in masters or doctoral level courses reflects the traditions of those fields, and their familiarity with different research approaches may vary a lot. This is a richness that can be used when designing a research method course for CER, but it is not obvious how to best exploit it. Moreover, most students from computing disciplines (and they were in our case a majority of the course participants) have little, if any, experience of qualitative research. The questions of trustworthiness of results from projects with small samples and interpretive approaches will almost inevitably bring about concerns among many students. Fourthly, the sheer extent of topics that could be covered in this cross-disciplinary field is very large. What should be preferred, and how deeply should various topics be covered? Should we teach methods as such, or aim at contextualizing all research?

We had initially no obvious answers to these questions. Moreover, we only found two previous courses with a similar content, one at Oslo University, Oslo, Norway<sup>5</sup>, and one at Uppsala University, Uppsala, Sweden<sup>6</sup>, to which we could look for examples. Therefore designing this course was rather an experiment than a straightforward pedagogical design task.

## 3 The course

The course was held in four intensive days during the extent of the full spring semester, and with substantial homework assignments to be done both individually and in groups. The workload was defined to be 8 ECTS credits or a bit over 5 weeks of full-time studies. The full course documentation, as well as the call, can be found at the course web site<sup>7</sup>.

### 3.1 The participants

Although 21 of the 23 participants were from Finland (two were from Estonia), they represented different levels of education (from masters students to PhD students very close to graduation) and with different educational backgrounds (computer science and engineering, of course, but also education and sociology). About half of the participants had actually been doing research in CER already; some were working in some other field of engineering education. But there were also a few participants who took the course from sheer interest in the topic or to get a course in research methods.

We, the teachers, also had mixed backgrounds: Anders Berglund is a senior lecturer in computer science, with his PhD in CER; Päivi Kinnunen is a PhD student in CER, with her licentiate degree in pedagogy, while Lauri Malmi is a full professor in computer science.

### 3.2 Aim and content

The course focused on pedagogically based research within CER and had a mainly methodological focus. The course aims are presented in Figure 1.

The content was thematized, with each theme corresponding to approximately one meeting:

**Meeting 1. Qualitative and quantitative research approaches.** Overview of qualitative and quantitative research, different research approaches, positivistic and interpretative research.

**Meeting 2. Results.** Presentations of some research approaches and their theoretical bases. Relationships between results and research approaches.

**Meeting 3. Quality.** Particularly trustworthiness in qualitative research.

<sup>5</sup> <http://www.ifi.uio.no/infdid/pensum.shtml>

<sup>6</sup> <http://www.it.uu.se/edu/course/homepage/datadidaktik/ht06?lang=eng>

<sup>7</sup> <http://www.cs.hut.fi/Studies/T-106.5550/english.html>. Clicking on "Wiki" shows all assignments.

#### Meeting 4. Participant presentations. Discussions on the quality of own projects.

Before each meeting the participants did assignments pertaining to the topic of the previous and/or next meeting. Participants were requested, for example, to read some articles that present qualitative/quantitative/mixed research approaches and then discuss the kind of results presented in the articles, how the results are related to the approaches, and the strengths and weakness of the papers. Another example is a preliminary phenomenographic analysis that the students were asked to do based on interview scripts they had been given. The most extensive individual assignment was a project that included an evaluation of the quality of a research project that the participants themselves had done, were doing at the moment, or would do in the future.

### 3.3 Rationale for the selected content

Deciding upon the content for a course within this subject area is a delicate issue. We had three rather diverse arguments for our choices.

The first is based in the nature of the regional CER community, which is dominated by the 'objectivist' (or 'positivist') perspective. Thus it seemed interesting to challenge this stand by problematizing what can be meant by a research result, how research can be performed, as well as the relationship between the two. The local environment is the source of the second argument: the selected course content was relevant for some of the PhD students at HUT. The third argument is pragmatic: the course content was to a large degree tailored to profit from the competence of the teachers.

We also discussed whether, and to what extent, the research methodologies should be contextualized in CER, or whether they should be presented in their own right without referring to their applications within CER. Our decision, for several reasons, was to emphasize the methodologies in a context.

Firstly, we did not want to restrict the content only to research methods, but we also wished to discuss

Computing Education Research. In this way, the course developed neither as a pure research methods course nor as a pure course discussing different theories and the outcomes of earlier studies. Secondly, we felt that this perspective would motivate the participants better, since it is focused on real cases. Thirdly, the versatility of issues that affect the choice of research methods and approaches is better highlighted through real life cases. Fourthly, we believe that the perspective we selected also enhances learning in a sense that it helps students to see the meaning and understand the relevance of not only research methods but also CER as a research area.

Our final argument goes outside the scope of the course: We believe, as Lister (2005) advocates, that it is important for CER to develop its own identity. This is, we believe, better done by putting the focus on research questions important to CER all through the course.

### 4 Lessons learned

This paper does not focus on the course results or learning outcomes even though the results of the course were quite satisfactory from our point of view (20 students have passed it until today). Rather we discuss our experiences of giving the course and our interpretation of the student evaluation, with the aim of identifying key points that we think should be considered when rearranging such a course.

The diversity of the participants in a cross-disciplinary field is a fact that needs to be taken into account. Students from different disciplines have various strengths, which is an advantage. When discussing and working together, students have the opportunity to learn from one another, to widen their perspectives and learn how to interact with peers who do not come from the same research tradition. However, the diversity brings along some challenges, too. Firstly, different disciplines do by their very nature emphasize different issues. Thus in a course such as ours the students' knowledge concerning research methodology may vary a great deal, from, for example, statistics and formal methods among computer scientists to data collection and qualitative research among educationalists. The diversity may also be one of the origins for the students' diverse motives to participate in the course. We also observed that the different cultures and values can lead to conflicts during the discussions. However, conflicts can be fertile starting points to the learning process if they are managed in a productive way.

The role of feedback for the participants needs to be highlighted. A research methods course such as this is more about developing new ways and critical thinking than about learning something specific. The students need to reflect on the newly acquired knowledge and new ways of perceiving things. This holds especially if the students are novices concerning the course topics. As the number of participants was more than twice what we had expected initially, we applied techniques where students presented their works in varying groups and discussed them together. However, feedback directly from teachers was clearly something that was expected more.

The issues concerning diversity and feedback led us to the following checklist.

After the course the student should

- have gained insights about quantitative and qualitative research approaches and their application,
- be familiar with some relevant research approaches,
- be able to select and apply an approach suited to a particular research question,
- be familiar with some methods for data collection and be able to judge the strengths and weaknesses in particular situations. Be able to select and apply suitable data collection methods, based on the research question and research approach, and.
- be able to design a study, to motivate the selection of a research approach and data

**Figure 1. Aims of the course Methods and results in computing education research, HUT, Finland, 2007**

*The participants and their backgrounds*

- Set clear prerequisites for the course to get more homogeneous students in relation to level of knowledge concerning research methodology, motives, and level of commitment.
- Students have different expectations of the course. Thus, make it clear what is expected from submissions on each assignment: how deeply should the topic be discussed in the submissions and how much are the students expected to refine the assignment?
- Harness the diversity of the students' backgrounds, for example, by asking students to give presentations concerning their expertise. Design exercises that require real collaboration with people from different fields.
- The different backgrounds of the students bring different ideas about what it means to learn something, what knowledge is, and how teaching ought to be performed.

*The giving of the course*

- Plan the course so that you can give feedback to the students and so that students get feedback from each other. Emphasize the quality of feedback so that it aids in developing students' thinking skills.
- Give hands-on exercises on data collection and analysis methods. However, the role of the theory should not be forgotten either. Otherwise hands-on exercises easily become a cookbook that is used without understanding the connection between research question, theory, and data collection and analysis methods.
- It might be profitable to concentrate on just a few research approaches and study them in depth.

*Organization of courses in CER*

- Offer several courses that discuss the research methodology and research in an area of CER on different levels.

Finally, according to our experience, we must be prepared to meet the limits of our own competence. As students have their own research goals, your experience may not cover those fields well enough.

## 5 Discussion questions

The diversity of CER is a source of confusion when offering courses of this kind. But we feel that we are addressing important issues. Therefore we hope to discuss the following questions within the conference presentation.

*How could we tackle the diversity of CER in a course of this kind?*

*Which topics should such a course include and what should the extent of the course(s) be?*

*How can we best exploit the skills of students with backgrounds from different disciplines?*

*How do we best design the course so that it really*

*reaches students with different backgrounds?*

*Which ways would be good to offer the students feedback?*

*To what extent is contextualization of research methods important in a course in CER?*

## Acknowledgement

We wish to thank Helsinki University of Technology for supporting the course through a grant. We also thank visiting lecturers Antti Rasila and Tony Clear for the valuable contributions to the course.

## 6 References

- Berglund, A. (2005). *Learning computer systems in a distributed project course: The what, why, how and where* (Vol. 62). Uppsala, Sweden: Acta Universitatis Upsaliensis.
- Berglund, A., & Lister, R. (2008). *Learning Educational Research Methods through Collaborative Research: The PhICER Model*. 10<sup>th</sup> Australasian Computing Education Conference, Wollongong, Australia.
- Fincher, S., & Petre, M. (2004). *Computer Science Education Research*. London, UK: Routledge Falmer.
- Fincher, S., & Tenenberg, J. (2006). Using Theory to Inform Capacity-Building: Bootstrapping Communities of Practice in Computer Science Education Research. *Journal of Engineering Education*, 95(4), 265 - 277.
- Lister, R. (2005). *Computer Science Teachers as Amateurs, Students and Researchers*. In proceedings of the 5th Baltic Sea Conference on Computing Education Research, Koli Calling. Keynote speech, Koli, Joensuu, Finland. 3 - 12.
- Pears, A., Seidman, S., Eney, C., Kinnunen, P., & Malmi, L. (2005). Constructing a core literature for computing education research. *ACM SIGCSE Bulletin*, 37(4), 152 - 161.
- Valentine, D. (2004). *CS educational research: a meta-analysis of SIGCSE technical symposium proceedings*. In proceedings of the 35th SIGCSE technical symposium on Computer science education Norfolk, Virginia, USA. 255 - 259.
- Wenger, E. (1998). *Communities of Practice. Learning, Meaning, and Identity*. Cambridge, UK: Cambridge University Press.

# Using Topic Map Technology in the Planning of Courses from the CS Knowledge Domain

**Evgeny A. Eremin**

Perm State Pedagogical University  
614990, Russia  
Sibirskaia str., 24, Perm  
eremin@pspu.ac.ru

## Abstract

A teacher must consider many details when planning and organizing a course. This paper shows how the new Topic Map computer technology can help in this process. The developed Topic Map for the CS knowledge domain is used for the analysis of introductory CS courses, but it can be also applied to other courses, including in non-computer disciplines. Well-structured Topic Maps may become a solid base in the formation of a common connected picture of learning science.

**Keywords:** Topic Map, XML, knowledge, computer, semantic, contents, structure, course, term, concept, relationship.

## 1 Introduction

Planning educational courses is one of the standard activities in teaching practice. Unification of learning contents in different institutions is impossible without using syllabuses and other governing documents that define the students' knowledge.

While the full knowledge domain is by its nature unified, its division into separate disciplines and other educational units raises the essential problem of planning. Interrelations between disciplines and topics bring additional pedagogical difficulties, but at the same time correctly organized repetition of material brings new possibilities to knowledge acquisition.

Traditionally all syllabuses are based on experts' experience, including attainments made by previous generations of educators. Despite some difficulties, educational planning is now well developed: Computing Curricula, created under the supervision of the ACM (ACM2001, ACM2005), presents an impressive example in the domain of computer science.

The present paper is devoted to the use of some new computer technologies in the planning process. Its main ideas are based on methods of knowledge representation, an area in which computer applications have recently made rapid progress. 'Electronic' knowledge

representation has proved useful for improving the development of courses and the systematization of their contents. Described proposals seem to be helpful in partitioning the course materials into lectures, prompting better timing and manner of introducing terms, accentuating associations between them, planning organized repetition, and so on.

The pedagogical aspect of the work consists in an attempt to create some reference point for planning CS educational courses. Shared and extended by colleagues, these results may in prospect lead to the creation of some machine-based specification of the CS educational knowledge domain.

Scrupulous development of the course structure is vital for the formation of a common connected picture of modern CS, which is divided into a set of separate courses. Students need some common outlook about internal computer logic, especially in the introductory stage. Some basic knowledge of that kind is required even in applied courses.

Similar ideas about the significance of structural knowledge in the physics domain were reported by Koponen, Mäntylä and Lavonen (2002). The authors even established a special course, whose "purpose is not to teach more physics but to organise what has already been learnt".

## 2 Development of the Topic Map

### 2.1 Problem definition

The main aim of the work was to develop a glossary for the general topic 'What is a computer'. By 'glossary' I mean a list of basic terms and the relations between them. As there are many concepts and names in the selected topic, the question of their choice is discussed later.

The vast number of interrelated terms (in knowledge presentation theory such a structure is often called a semantic net) makes it difficult to render visually. For example, a graphic image of the scheme, usually suitable for small groups of concepts, becomes unreadable for a large knowledge domain. So the Topic Map technology (see reference TopicMap) was applied to form the required complex glossary. With this technology, all data is stored in XML format, and so can be processed afterwards in different ways.

Free **Topic Map for Learning** software (Dicheva and Dichev 2007, reference TM4l for download) was used to

build the glossary. It includes a powerful Topic Map editor accompanied by a specialized viewer.

The Topic Map structure supports the implementation of links to different kinds of resource, but this important feature has not yet been put to use in this work.

## 2.2 Knowledge Domain

As follows from the problem definition, the selection of terms for the glossary has great importance. On one hand, the knowledge domain we analyse is general and broad, so we must include as many glossary components as possible. On the other, the review of an excessively enlarged list may become awkward. So in this work the selection of terms for the glossary obeys the following principle: only general terms with essential interrelations, which form the kernel of the 'Computer' topic, were accepted; particular terms, simply naming the objects, were not considered to make the glossary any clearer. For example the terms 'input device' and 'output device' are essential for the common picture, but concrete device names ('scanner', 'keyboard' etc) are considered to be details. Similarly 'operating system' and 'file system' were included in the domain, but their realizations ('Windows', 'Linux', 'NTFS', 'Ext3') were not. Note that terms in the first category change rather less frequently than those in the second.

The total set of concepts in the resulting glossary consists of more than 120 terms. They can be easily seen in the

Relationship	Examples
whole/part	processor is assembled on main board; instruction consists of clock cycles; data and software are stored on medium
class/subclass	kinds of memory: RAM, external etc; RAM can be static and dynamic; address counter is a register
class/instance	text editors: Word and TeX; keyboard and scanner are input devices
sense	machine word-processor, capacity-byte
analogy	program and hardware interface
base	data coding is fully based on binary system; main processor algorithm necessarily uses address counter
use	interaction between units uses bus; addressing is <i>one of</i> the principles of memory
characteristic	capacity characterizes processor and memory cell; processor is characterized by clock frequency
realization	ALU realizes arithmetic and logic operations; program processes data; ROM BIOS actualizes booting
control	operating system controls hardware; controller governs I/O device
connection	controller may be connected to the extension slot; plug connects external device to system unit

**Table 1: Relation types**

TM4I window as a list of topics.

## 2.3 Knowledge Representation

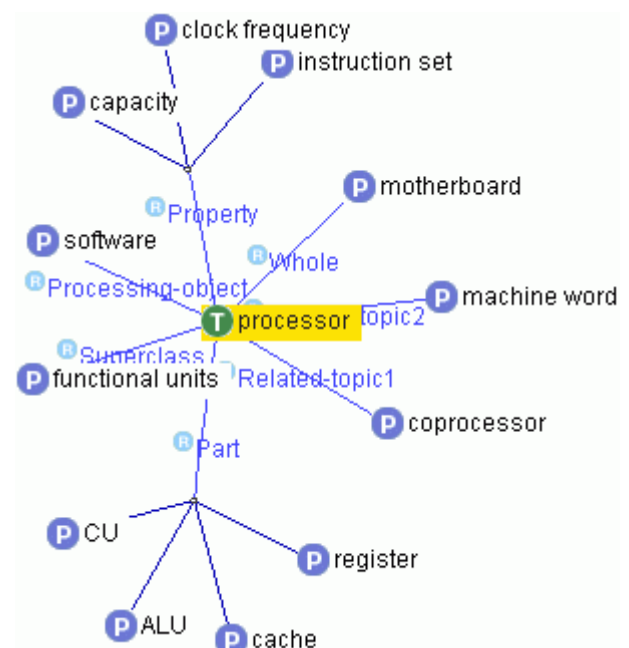
The traditional theoretical representation of knowledge (including, by the way, the Prolog language) is by way of pair relations. In creating the glossary, objects were also formed into pairs according to their relations in educational material. For purposes of identification, each relationship gets its own name. For example the notion in human language 'a processor is the functional unit of a computer' translates as follows: object 'functional unit' – relationship 'class/subclass' – object 'processor'.

Although in reality there are many different kinds of relation between objects, it has proved possible to represent all of the knowledge domain by means of a limited number of common relationships. Their full list is shown in table 1.

The table represents all of the relationships used in the glossary; to illustrate the breadth of the terminology, several different examples are given for each. It can be clearly seen that the first five relations are universal and conventional (see Booch 1993 for example); 'base' and 'use' are used mainly for linking with theory; and the last relations reflect some specifics of the CS knowledge domain and are encountered more rarely than the previous ones.

Let us discuss an example of using the formalism described above. We can formulate the following important notions for the object 'processor'.

- *Clock frequency* and *capacity* characterize a *processor*.
- Every *processor* has its own *instruction set*.
- *Processor* is a computer *functional unit*.
- *Processor* executes *software programs*.
- *Processor* is assembled on the *motherboard*.
- A *processor* for specific functions (for example



**Figure 1: Relations for the object 'Processor'**

mathematics, graphics) is named a *coprocessor*.

- *Machine word* is a unit of data with which *processor* works.
- The main parts of the *processor* are *ALU* and *CU*. Modern *processor* contains *cache*.
- *Processor* uses *registers* in its functioning.

After we convert the above list into Topic Map pairs, we get the picture presented in figure 1, a screenshot from the TM4I viewer).

All other terms in the glossary are processed the same way, leading to the full Topic Map for the 'Computer' knowledge domain. The contents for this formalization were chosen in accordance with computer architecture and organization textbooks (for example Tanenbaum 1998, Hamacher, Vranesic and Zaky 2001).

As we haven't used any specifics of CS except in the data for the Topic Map, the described technology is universal and can be applied to other disciplines and topics.

### 3 Discussion of results

The original pedagogical motive for building a glossary of CS basics arose as an attempt to make computer courses for novices more thoughtful. Computers are so widespread in everyday life that many people must be able to use them efficiently. The conventional solution to the problem (at least in Russia) is so-called user courses, where students acquire computer literacy. Such courses are usually very sketchy and don't provide the required level of computer control. The main reason, it seems to us, is a lack of understanding of computer logic. For example such 'intuitive' users can easily lose files after saving because they have too primitive notions (or no notions at all) about file systems. So including fundamental CS concepts into introductory courses is desirable.

Modern educators often pay too little attention to this aspect, although serious pedagogical research gives evidence of its importance. For example duBoulay (1989) wrote that "even if no effort is made to present a view of what is going on 'inside' the learners will form their own". The review article of Ben-Ari (2001) thoroughly argues that some theoretical knowledge about computer organization "must be *explicitly* taught and discussed, not left to haphazard construction and not glossed over with facile analogies". It must be done even for those students who study office software (many of Ben-Ari's examples describe the learning of MS Word).

Such background makes an educational analysis of CS fundamentals very timely. The availability of reference points in the form of a glossary may prove to be helpful for planning different computer courses.

Let us analyse the glossary developed in this work and see how it can help in the choice of computer course material.

First of all, relationships between CS concepts show that user courses, being aimed mainly at software applications, are distinct from the CS foundation. For confirmation we may retrace the relations for binary

system, for example. The Topic Map gives us the following chains of terms: binary system – binary coding principle – coding of data and instructions – instruction set and coding of numbers, texts etc. So if coding of data and instructions is not considered in computer education, learning of the binary system becomes isolated. We can get similar results for logical elements and circuits.

From the other side, the glossary clearly shows which topics are likely to cause difficulty for students who have studied only a typical office course. The glossary predicts that such students lack knowledge in hardware control by means of an operating system, program loading, file system, data representation, and so on.

The developed Topic Map may be recommended as a support tool in educational planning in one more way: it can bring to light the concepts that pass through several distinct topics. Such cohesive terms must certainly be emphasized in CS course material. The example of one such term, 'byte', is presented in figure 2, demonstrating the visualization possibilities of the TM4I Editor.

Examination of the picture shows that the term 'byte' has associations with different topics of several areas, and so this concept must be carefully formed in students' minds. A byte in CS courses is:

- a unit of measure of information;
- closely connected with bit and other units;
- the minimal addressing part of RAM (in fact its cell);
- used for characterizing the capacity of a processor (machine word).

Experienced lecturers know how important it is to explain the difference between the minimal discrete unit, 'bit', and the minimal *addressing* part of memory, 'byte'. Relations with storage of different data types, seen in figure 2, are also important.

The numbers on this figure mean the number of hidden

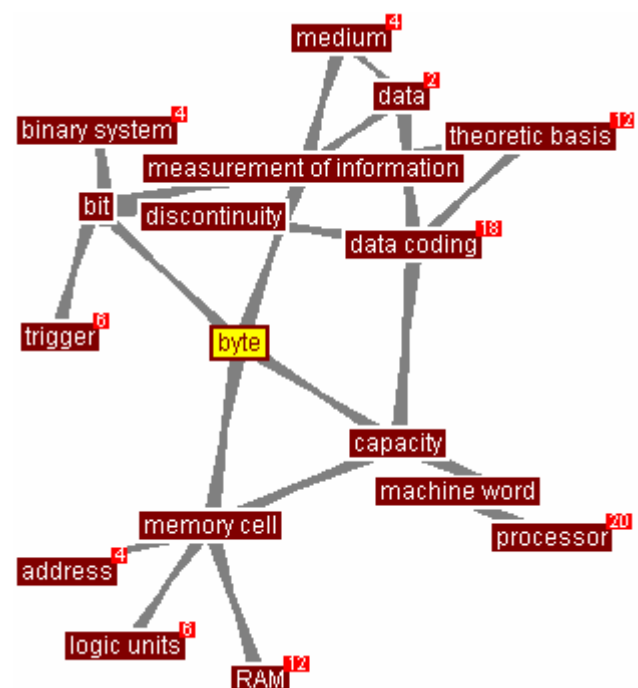


Figure 2: Role of the term 'byte' in the course

graph nodes (topic names), which the software hasn't depicted on the screen. These hidden terms can be displayed in TM4I just by a mouse click.

It is necessary to emphasize the difference between figure 1 and figure 2. The first one (generated by the TM4I viewer) presents the relations of the single object 'processor'. The second image (produced by the visualizing tool of the TM4I editor) tries to demonstrate *all* relations in the knowledge domain, with the topic 'byte' in the centre and a radius of 2 objects (radius can be set arbitrarily in the TM4I software). Note that unlike figure 1, figure 2 doesn't give us the option of identifying relationship types.

Sceptics may say that a Topic Map, being a form of expert knowledge base, will show nothing beyond what the experienced lecturer already knows. But we must keep in mind that in processing large knowledge bases the computer has some advantages – it never misses any detail. The visual form of knowledge representation must also be mentioned: without this, teachers would need much more time and effort to show their own schemes to students and fellow teachers.

Because Topic Maps are saved as computer files in the XML standard, they are not limited to the concrete form of data presentation shown here. If some new software appears, for example, generating 3D-views of Topic Maps, it can be immediately applied to our map. This feature is an essential difference between Topic Maps and Concept Maps and other similar diagrams. For instance, Concept Maps, drawn with specialized software, can also represent interrelations inside a knowledge domain. But it is clear that graphic image in Concept Map is the primary structure, and hence is fixed in form. By contrast, a Topic Map is simply text information, and any graphic rendering can be applied to it later.

The Topic Map of the computer domain created in this work is itself some kind of result which can be shared with colleagues. The file with the resulting map can be downloaded from the Internet (see link tmComputer in the references). The offered Topic Map can be easily extended and modified by means of the TM4I editor.

The technology of building an 'electronic equivalent' of the contents of a course, described in the paper, is universal and may be applied to the planning of other courses, including non-computer ones.

#### 4 Acknowledgement

Special thanks to my colleague M.T. Sharov for long and thoughtful discussions about hardware details and their interrelations. These discussions significantly improved some branches of the developed glossary.

#### 5 References

- ACM2001: Final report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science.  
<http://www.acm.org/education/curricula.html#cc2001-fr>. Accessed 3 Dec. 2007.
- ACM2005: Computing Curricula: 2005 Overview report.  
<http://www.acm.org/education/curricula.html#CC2005>. Accessed 3 Dec. 2007.
- Ben-Ari, M. (2001): Constructivism in Computer Science education. *Journal of Computers in Mathematics and Science Teaching* **20**(1):45-73
- Booch, G. (1993): *Object-oriented analysis and design with applications*. Menlo Park, CA, Addison-Wesley.
- Dicheva, D. and Dichev C. (2007): Authors support in the TM4L environment. *Information Technologies and Knowledge* **1**(3):215-218.
- du Boulay, B. (1989). Some difficulties of learning to program. In *Studying the novice programmer*. 283-299. E. Soloway and J.C. Spohrer (eds.). Hillsdale, NJ, Lawrence Erlbaum.
- Hamacher, C., Vranesic, Z. and Zaky, S. (2001): *Computer organization*. New York, McGraw-Hill.
- Koponen, I.T., Mäntylä, T. and Lavonen, J. (2002): Challenges of Web-based education in physic teachers' training. *Proc. of International Conference on Information and Communication Technologies in Education*, Badajoz, Spain, 291-295.
- Tanenbaum, A.S. (1998): *Structured computer organization*. Upper Saddle River, NJ, Prentice Hall.
- TM4I: Topic Maps 4 e-learning.  
<http://compsci.wssu.edu/iis/nsdl/>. Accessed 3 Dec. 2007.
- tmComputer: Topic Map for Computer Domain.  
<http://www.pspu.ru/personal/eremin/eng/tm/tmc.html>. Accessed 3 Dec. 2007.
- TopicMap: ISO/IEC 13250:2002 Topic Maps.  
[http://www.y12.doe.gov/sgml/sc34/document/0322\\_files/iso13250-2nd-ed-v2.pdf](http://www.y12.doe.gov/sgml/sc34/document/0322_files/iso13250-2nd-ed-v2.pdf). Accessed 3 Dec. 2007.

# Exploration Module for Understanding the Functionality of the Internet in Secondary Education

Stefan Freischlad

Universität Siegen  
Didactics of Informatics and E-Learning,  
Hölderlinstraße 3, 57068 Siegen,  
freischlad@die.informatik.uni-siegen.de

## Abstract

In developing and successfully implementing a classroom project on ‘Internetworking’ in secondary education, the author found that there is a need for adequate educational software. Abstract informatics concepts have to be connected to real-life experiences. In this paper the concept of views that support different perspectives on the subject is applied to meet the requirements for educational software in secondary education in terms of activities and abstraction. The educational software developed allows analysis, simulation, and construction of network infrastructures and connects the visible behaviour, inner structure, and implementation details of Internet applications. Thus it provides a motivating approach based on the concept of discovery learning.

*Keywords:* educational software, exploration, Internet applications

## 1 Learning About the Internet in Secondary Education

In June 2005 we started a research project, promoted by the German Research Foundation (DFG), with the aim of analysing the requirements of informatics education and developing a didactic concept in the field of ‘Internetworking’ for secondary education. Although there is a broad consensus that knowledge about the Internet has to be part of informatics education (Tucker 2003, van Weert 2000), we found that there is no relevant science-based didactic concept for secondary education. Therefore we are developing the didactic system ‘Internetworking’, which comprises didactic concepts and learning material. In developing and successfully implementing classroom projects about Internetworking in secondary education, the author discovered the need for practice-oriented learning material. The research question in this paper is how learning material about ‘Internetworking’ can be designed based on the constructivist learning theory.

Kurose and Ross (2005) describe their approach for higher-education teaching about the Internet by starting with applications. They give three reasons for applying a top-down approach to the Internet layer model. First, students are really motivated and interested in how the Internet works if they start with the applications and the application layer and then proceed in more detail through underlying lay-

ers. The second argument is the exceptional meaning of the application layer for actual developments. And the third reason is that it is possible to introduce network application programming earlier in the course. In fact, linking both aspects – informatics concepts and applications – is necessary for the understanding of informatics systems (Stechert 2007). Therefore, practice-oriented learning material that bridges the gap between abstract informatics concepts and Internet applications is necessary for informatics courses in secondary education. Classroom activities would start with a motivating example of an Internet application or another phenomenon within the scope of our project.

We started analysing the characteristics of Internet-based informatics systems to identify the necessary competencies for using Internet applications (Freischlad 2006) and described our theoretical approach towards three components of the didactic system ‘Internetworking’. The ‘knowledge network’ describes the learning objectives and necessary prerequisites of learners. ‘Exercise classes’ represent the abstraction of specific exercise contexts and are hierarchically structured (Freischlad and Schubert 2007). The third component is ‘learning aids’, particularly educational software. We developed learning materials based on the didactic system and implemented them into classroom practice, that is, we engaged in curriculum intervention. The first project was primarily about communication and privacy protection on the Internet. The second project focused on Internet structures. And the third project combined several aspects of the preceding projects. The results of the projects led to modification and refinement of the didactic system. The curriculum intervention brought us valuable information about students’ prior knowledge, attitudes, and misconceptions. At the same time we have started the development of the educational software for ‘Internetworking’ as a student project, which we call FILIUS<sup>1</sup>. This software is required for practical exercises and experiments.

## 2 How to Understand Internet Applications and Services

### 2.1 Understanding of Informatics Systems

According to Stechert (2007) the analysis of characteristics of informatics systems must focus on the visible behaviour (A), the inner structure of a system (B), and specific internal details (C). Therefore, educational software that fosters understanding of the functionality of Internet applications has to consider these perspectives. The visible behaviour is represented by user agents such as web browsers and email

Copyright ©2008, Australian Computer Society, Inc. This paper appeared at 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. Raymond Lister and Simon, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

<sup>1</sup> “Freie Interaktive Lernumgebung für Internetworking der Universität Siegen”  
(URL: <http://www.die.informatik.uni-siegen.de/pgfilius/>)

software, but it doesn't include data exchange with protocols. The inner structure is composed of several informatics concepts, for example, protocols as sets of rules for data exchange, and the format and meaning of data units that are exchanged between the entities on different hosts. Specific internal details are represented, for example, by the source code of programs.

Brinda (2006) describes how the concept of exploratory discovery learning was applied to learning with informatics systems. Informatics experiments are identified as a starting point. He then derives requirements for 'discovery friendly application software', which must support exploration and experiment. He concludes that 'discovery friendly learning' must provide motivating learning activities, the possibility of continuation, and linking of construction and simulation. Therefore, when it is introduced for discovery learning, educational software must allow a high level of interactivity, comprising the engagement categories of Viewing, Changing, and Constructing according to the engagement taxonomy of Naps et al. (2002).

Bruner (1966) specifies three representations that are used within the learning process. "At first the child's world is known to him principally by the habitual actions he uses for coping with it. In time there is added a technique of representation through imagery that is relatively free of action. Gradually there is added a new and powerful method of translating action and image into language, providing still a third system of representation" (Bruner 1966, p. 1). He calls these modes of representation 'enactive', 'ikonic' and 'symbolic'. Even though these representations are developed step by step in a child's cognitive development, they are important not just for adolescents but also for adults: "their interplay persists as one of the major features of adult intellectual life" (Bruner 1966, p. 1). Jank and Meyer (2005) explain that even though these representations were developed step by step they were kept in learners' minds throughout their lives. Furthermore, complex mental representations require integration of the three modes – including the enactive mode which is defined by learners' activities. They conclude that the interplay between cognition and activities is integral to the development of mental representation, and therefore to learning (Jank and Meyer 2005, p. 322). Synchronisation of activities with ikonic and symbolic representations provides students with the means for this interplay.

## 2.2 Related Works

There are already approaches published that aim at learning about informatics concepts linked to the Internet. We implemented a concept within a course at university. In order to provide an enactive approach to the issue of computer networks in the course 'Didactics of Informatics', students had to build a local area network with six notebooks and one hub. They had to configure the notebooks and test their configuration with simple tools like ping or a tool for sending messages to other notebooks. Furthermore, they integrated the notebooks into a user domain. Students were very motivated because they could test their configuration with applications and we provided the students with the output of Wireshark<sup>2</sup> on a whiteboard. This hands-on approach is not feasible in those schools where we conducted curriculum intervention. These schools have no notebooks that can be used for this purpose in informatics courses, because learners would need administrative privileges for network

configuration. Corbesero (2003) describes a similar concept for higher education.

Steinkamp (1999) analysed how discovery learning with experiments could be transferred from the natural sciences to informatics. As a prototype for necessary interactive learning material he developed an exploration module for computer networks. The software makes it possible to build computer networks out of different components. After the network is constructed learners can test their environment with a web browser. A second view allows the observation of status messages which can be used to analyse the situation. The components comprise both hardware and software, for example, DNS server, web server and web browser, Internet Protocol, socket, switch, and the abstract component Internet represented by a cloud. These components are connected on the desktop. With this prototype Steinkamp showed how experiments could be performed in informatics education. But there are several restrictions linked to this software. Software and hardware components are not distinguished. Therefore, students cannot see that several services are processes on the same hardware. The abstract component Internet represented as a cloud does not explain the characteristics of the Internet. And it is not possible to configure web services.

Kornelsen et al. (2005) describe an educational software package for discovery learning about Internet services. The concept is based on a generic framework with a layered architecture composed of the three layers user interface, service broker, and Internet services. The user interface provides views for input of requests for Internet services and for representation of received answers. This concept allows extensibility with illustrative and learner specific views, but it provides no complete picture of the Internet. Services are not explored within the application context. Therefore the visible behaviour and the inner structure of the informatics system are not linked. Besides, it is restricted to Internet services at the application layer.

A third approach is the use of virtual machines. It supports the facility to run several machines on the same computer. Each host can be configured and any software can be used. Students work within a familiar environment with original versions of applications. A network analyser such as Wireshark can be used to observe the data exchange. Preconfigured virtual machines are used for given scenarios. The realisation in school practice is limited by problems with hardware and license restrictions. Furthermore, this approach does not provide an ikonic representation of the computer network.

## 3 Concept of the Educational Software

### 3.1 View Conception

From the need for students to attain an overall picture of the subject, Brinda (2006) concludes that exploratory software should provide different views on it. While students are familiar with graphical user interfaces of Internet applications, these interfaces hide the (physical) structure of the Internet and data exchange between the components, which are important aspects of the overall picture of Internet applications. The aim of FILIUS is to support students in understanding the functionality of the Internet and applying it. Therefore, the following items have to be considered:

- the items A-C describing characteristics of an informatics system to give students an overall picture;

<sup>2</sup>see <http://www.wireshark.org>

- construction and simulation to support discovery learning;
- variation of program execution through parameters and modification of the structure of the 'world';
- views assigned to the enactive, ikonik and symbolic modes of representation;
- synchronisation of activities with views providing ikonik and symbolic representations; and
- persistent saving of scenarios for continuation.

We defined four views for the realisation of these requirements. The 'Network View' (I) makes it possible for students to construct local area networks and interconnected networks out of hardware components using a visual representation. These components can also be configured with parameters, for example, the IP addresses of hosts. Connections between hardware components are highlighted during data exchange. The 'Operation View' (II) makes available the virtual graphical user interfaces of hosts, making it possible for users to install or remove software and to start, configure, and use the applications. The 'Message View' (III) allows the observation of data units that are exchanged between hardware components on different layers of the Internet protocol stack, and shows the status messages of selectable components. The 'Code View' (IV) enables students to develop and implement applications of their own or to manipulate these software elements. It provides an editor for source code amendment and manipulation and the facility to compile these programs for direct use.

With these different views learners are able to discover implications caused by changes or manipulations in one view or another. One example is the effect when cutting the connection between two local networks while just one network includes a domain name server. In the network without a domain name server it is no longer possible to retrieve a web page from a web server by its domain name, because the domain name cannot be resolved. Students can analyse this situation by observing the data exchange between the components.

### 3.2 FILIUS – a Practice-Oriented Approach

FILIUS comprises basic generic functions and two different modes: a design mode for the construction of computer networks and software and a simulation mode for the simulation of Internet applications. The basic functionality involves persistent saving of scenarios comprising computer networks, configurations, and applications. Therefore, it is possible for learners to analyse, extend, modify, or simulate given networks. Additionally, it is possible to compare student solutions in the classroom to explain the solution to a given task.

The design mode enables students to construct, extend or modify computer networks. Hosts, switches, routers, and connections are available to construct local and interconnected networks. Additionally, there is a Modem component, which allows the connection of two instances of this software to build up a larger network with components on different computers. It is possible to configure the components in design mode. Each host must be configured either manually, by explicitly configuring IP address, network mask, and default gateway, or by dynamic host configuration protocol (DHCP) if the local network includes a server that supports DHCP. Each host can provide a DHCP server. Furthermore, advanced students can extend a generic client-server application

and modify given components. The simulation mode offers a view of the entire network. A virtual desktop can be opened for each host. The configuration of each host can be modified by installing or removing applications. Several generic tools such as a file explorer and Internet applications and services are supported. Each application provides a simple graphical user interface for configuration and use or starting and stopping of services. Data exchange can be observed within the network representation, with connections highlighted during data exchange and transferred messages shown within a message dialogue box similar to network analysis tools such as Wireshark.

The Internet protocol stack is implemented to ensure that students can do their own experiments. The application layer is implemented with several Internet applications, that is, email, World Wide Web, and peer-to-peer software for file sharing, and Internet services, that is, DNS and DHCP. The transport layer comprises transmission control protocol (TCP), which is used for Internet applications, and user datagram protocol (UDP), which is used for the services DNS and DHCP. The implementation of the Internet layer comprises datagram forwarding with static forwarding tables that are part of the router. Dynamic routing is not supported. This layer also includes the address resolution protocol (ARP), which is necessary for data exchange on local area networks based on IP addressing. The underlying subnet layer is implemented as a simple protocol.

### 4 Activities of Students

Freischlad and Schubert (2007) describe a hierarchical classification of exercise classes, derived from an analysis of textbooks, for the didactic system 'Inter-networking'. Apart from previous knowledge about computer networks and information security they discern five classes at the first level which are concretised at the second level. The first major class is 'Applications', and includes their characteristics, their uses, and their specific realisations as distributed systems on the Internet. The other classes are designed to foster understanding of the functionality of these applications. The 'Protocols' class also includes Internet security because Internet-specific security mechanisms are implemented through protocols. The 'Addressing' class comprises identification of hosts within computer networks. Additionally, this class includes directory services that support information about an address. The 'Data Transfer' class describes what is going on when data has left one end-system and before it reaches the targeted end-system. The last class, 'Architecture', is composed of models and informatics systems that are essential for the architecture of the Internet, considering both hardware and software.

The approach of Stechert and Schubert (2007) for the learning process starts with the analysis of an informatics system. Brinda (2006) demands construction and simulation for discovery learning. Therefore we discern the three activities analysis, simulation (including modification of parameters), and construction. Table 1 displays the supported activities depending on the exercise classes. In the first column the observable characteristics of informatics systems are presented, while in the second and third columns the corresponding views are listed. Applications are constructed in the Code View. The Operation View allows users to manipulate parameters, for example, email configuration or DNS records. The graphical user interface or the Operation View allows observation of visible behaviour, and the Code View allows analysis of implementation details. Protocols

	Analysis	Simulation	Construction
Application	A, C	Operation	Code
Protocols	B	Code	Code
Addressing	A, B	Network	Network
Data Transfer	B	Network	Network
Architecture	A, C	Network, Operation	Network, Operation

Table 1: Support of Student Activities (A is visible behaviour, B is inner structure, C is specific internal details)

of the application layer are constructed and parameters are modified in the Code View. Protocols are not part of the visible behaviour of an informatics system. Therefore, students can analyse the internal structure just by means of the Message View and implementation details via the Code View. The Network View supports construction and modification of networks, that is, Addressing, forwarding (which is part of Data Transfer), and interconnected networks (Architecture). Overlay networks that are necessary for peer-to-peer file sharing are constructed and modified in Operation View. They are also part of the exercise class Architecture. Thus every exercise and activity is supported, while Applications, Addressing, and Architecture are emphasised in consideration of their role in the understanding of informatics systems.

## 5 Conclusions

The educational software enriches classroom practice, such as the learning process about how local area networks are built up, with student activities. Without this software we required an analytical approach, because it was not possible for students to configure or modify components. This software enables students to build up their own computer network out of different components, which must be configured. After building a network they can test it with applications such as email or WWW. Having simulated a computer network, students can validate their hypotheses by testing its functionality through Internet applications.

The different views of the learning software enable students to acquire an overall picture of selected informatics systems, such as email. For this purpose, the author proposes that learners build up an entire infrastructure with at least two email servers. Furthermore, they should include two computers, each with email software, so that different users can send mails to each other. A DNS server is needed for domain name resolving. Thus, learners are able to connect the user-visible behaviour and the inner structure of the informatics system, the latter being represented by the static view of the computer network and the dynamic view of exchanged data.

## 6 Acknowledgements

The author gratefully thanks the members of the student project team FILIUS: André Asschoff, Johannes Bade, Carsten Dittich, Thomas Gerding, Nadja Haßler, Johannes Klebert, and Michell Weyer.

## References

- Brinda, T. (2006), *Discovery Learning of Object-Oriented Modelling with Exploration Modules in Secondary Informatics Education*, Springer Science, Educ Inf Technol (2006) 11: 105-119.
- Bruner, J. S.; Olver, R. R.; Greenfield, P. M. (1966), *Studies in Cognitive Growth*, John Wiley, New York.
- Corbesero, S. G. (2003), *Teaching System and Network Administration in a Small College Environment*, in JCSC 19, 2 (December 2003), pp. 155-163.
- Freischlad, S. (2006), *Learning Media Competences in Informatics*. in *Proceedings of Second International Conference on "Informatics in Secondary Schools. Evolution and Perspectives - ISSEP"*, Vilnius, Lithuania, pp. 591-599.
- Freischlad, S.; Schubert, S. (2007), *Towards High Quality Exercise Classes for Internetworking*, in *Proceedings of IFIP-Conference "Informatics, Mathematics and ICT (IMICT2007): A golden triangle"*. Boston, USA.
- Jank, W.; Meyer, H. (2005), *Didaktische Modelle*, Cornelsen, Berlin.
- Kornelsen, L.; Lucke, U.; Tavangarian, D. (2005), *Expedition in das Datenreich: Exploratives Erlernen von Internetdiensten*, in Haake, J. M.; Lucke, U.; Tavangarian, D., eds., *Proceedings of DeLFI 2005*, LNI, GI e. V., Rostock, pp. 271-282.
- Kurose, J. F.; Ross, K. W. (2005), *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, Amsterdam.
- Naps, T. L.; Rößling, G.; Almstrum, V.; Dann, W.; Fleischer, R.; Hundhausen, C.; Korhonen, A.; Malmi, L.; McNally, M.; Rodger, S.; and Velazquez-Iturbide, J. A. (2002), *Exploring the role of visualization and engagement in computer science education*. In *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education*.
- Stechert, P. (2007), *Understanding of Informatics Systems - A theoretical framework implying levels of competence*, in A. Berglund; M. Wiggberg, eds., *Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling*.
- Stechert, P.; Schubert, S. (2007), *A Strategy to Structure the Learning Process Towards Understanding of Informatics Systems*, in *Proceedings of IFIP-Conference "Informatics, Mathematics and ICT (IMICT2007): A golden triangle"*. Boston, USA.
- Steinkamp, D. (1999), *Informatik-Experimente im Schullabor*, Diploma thesis, University of Dortmund, Faculty of Informatics, URL: <http://www.die.informatik.uni-siegen.de/lehre/diplom/steinkamp/>
- Tucker, A., ed., (2003), *A model curriculum for K-12 computer science: final report of the ACM K-12 task force curriculum committee*, ACM, New York.
- van Weert, T., ed., (2000), *Information and communication technology in secondary education - A Curriculum for Schools*, UNESCO, Paris.

# Should we assess our students' attitudes?

Ursula Fuller and Bob Keim

Computing Laboratory  
University of Kent  
Canterbury UK

U.D.Fuller@kent.ac.uk – R.G.Keim@kent.ac.uk

## Abstract

Professional skills and attitudes form an increasingly large part of the requirements of computer science graduates. Students are assessed on their knowledge and cognitive skills but not on the attitudes that will lead them to practice in the workplace what they have been taught in the classroom. This paper argues that it is necessary to assess attitudes as well as cognitive achievements and that this is feasible within existing curricula.

**Keywords:** Affective domain, Professional issues.

## 1 Introduction

The good CS student is one who believes the subject is valuable and exciting, who has acquired an instinctive feel for what constitutes an elegant solution to a software engineering problem, and who has incorporated professional values into her or his world view. This student's achievement, in other words, is characterized by attitudes and values as much as by knowledge and cognitive skills. By contrast, the learning outcomes of such a student's programme of study are likely to be expressed entirely in cognitive terms and the assessment tasks will have been framed to focus on aspects of the learner that can be measured 'objectively' and without judging the student's value set. There is a fundamental lack of constructive alignment between the values that CS educators hold dear as reflecting the essence of their subject and the way in which they assess their students.

The discipline of computing has been described as one that "combines the ethos of the scholar with that of the professional; it underpins the development of both small-scale and large-scale systems that support organizational goals" (QAAHE 2000). To become a computer scientist is to accept this ethos as much as to master a prescribed set of knowledge and skills. To measure whether a student has done so requires assessment of attitudes and values, in other words, assessment in the affective domain. This paper argues that we short-change our students if we do not do so and that such assessment is feasible within existing curriculum frameworks.

## 2 What is the Affective Domain?

Learning objectives can be divided into three domains, *cognitive*, *affective* and *psychomotor* (Bloom et al. 1956).

The cognitive domain is concerned with objectives involving intellectual activities ranging from remembering and applying to analyzing, synthesizing and evaluating. The affective domain deals with objectives involving emotions or a degree of acceptance or rejection, which may be expressed as interests, attitudes and values. Although learners respond holistically, there is little correlation between cognitive achievement and attitudes and values, so that achievement in one domain cannot be used to predict achievement in another.

Krathwohl et al (1964) proposed a five-level taxonomy of the affective domain as follows:

1. **Receiving** The learner is aware of the topic and is willing to learn about it.
2. **Responding** This ranges from reluctant compliance, via willing response, to a sense of satisfaction in doing what is required.
3. **Valuing** The learner ascribes worth to the topic, believes in it and is committed to it.
4. **Organization** The learner organizes a set of values into a value system that can be used to respond to situations.
5. **Characterization by a value or value complex** The learner has completely internalized the values to the point at which they characterize the individual.

They noted that there was very little assessment in the affective domain for a number of reasons. Firstly, it is much harder to measure performance in the affective domain, partly because there are very few validated assessment instruments but also because students can tell which responses will be rewarded or penalized and answer accordingly. For example, a student can profess commitment to honesty while at the same time handing in plagiarized work. If the latter is undetected, the student could get a good mark for academic integrity.

A second difficulty with the assessment of affect is that values and attitudes are seen as private matters. Students should be free to hold their own opinions, regardless of those of their teachers. An instructor who hopes to inspire students with a love of database technology will not lower the mark for an excellent piece of data modelling because the student adds at the end "I hate databases". Responses in the affective domain are not unidirectional; a student who is indifferent to databases at the start of the course may love or hate them by the end. By contrast, development in the cognitive domain is always positive.

Finally, affective behaviour changes at a much less predictable rate than cognitive behaviour. Some students may not really realize the importance of testing until the

final year; on the other hand, they may develop a lasting enthusiasm for a topic the first time they encounter it.

### 3 Values in the CS Curriculum

The ITiCSE'99 Working Group on Professionalism pointed out that graduation with a diploma in the field is the major indicator of competency in ICT and therefore, because belonging to a professional body is not a prerequisite for working as a computing professional, educators need to take the development of professional qualities very seriously in designing their curricula (Little et al. 1999). The working group identified a set of characteristics of a computing professional, such as "Shows a personal commitment to quality", "Understands and thinks like those they serve", "Takes pride in work" and "Reaches out for responsibility". These go far beyond the Professional Practice curriculum area. Instructors delivering any course in the computing curriculum that involves the development of software artefacts, be they Java programs, a complex spreadsheet or a set of word processing stylesheets and macros, will be seeking to instil such values. They will be happy if they find that their ex-students have not only grasped some theory of software quality but are continuing to apply it as they use computers after graduation.

The Computing Curricula 2001 (CC2001), drawn up by the IEEE and ACM, have so much influence on the content of CS degrees worldwide that this paper uses them as an exemplar. CC2001 is focused mainly on the cognitive domain, but refers also to the values and attitudes that computing students should acquire. Some of these references are explicit and others are implicit.

The Joint Task Force on Computing Curricula (2001) includes "the importance of testing" in the concepts to be covered in the introductory curriculum. This cannot be intended to be value-free. It is unthinkable that an assignment for a Computer Fundamentals course could require students to discuss whether testing was or was not important, and award a good grade to an essay that concluded that it was not. What is needed, and implicitly expected by CC2001, is that students should believe that testing is important and that they should routinely act on that belief whenever they work on a piece of software.

CC2001's specification of the general characteristics of CS graduates includes the headings: *System-level perspective*, *Appreciation of the interplay between theory and practice* and *Familiarity with common themes*, which are then explained using phrases such as "encompass an appreciation for", "recognize that these themes have broad application ... and must not compartmentalize them". These suggest that the Joint Task Force's approach to concepts and skills is to require active, enquiring engagement rather than passive awareness. Under the heading Adaptability, the CS curriculum says

Fundamentally, teaching students to cope with change requires instilling in those students an attitude that promotes continued study throughout a career. [...]

[CS graduates should] keep abreast of current developments in the discipline to continue [their] own professional development.

The detailed list of capabilities and skills for computer science graduates in CC2001 includes:

Produce work involving problem identification, analysis, design, and development of a software system, along with appropriate documentation. The work must show some problem-solving and evaluation skills drawing on some supporting evidence and demonstrate a requisite understanding of and appreciation for quality.

This is in accord with the ITiCSE'99 Working Group's characterization of a computing professional and involves an affective response, actively valuing quality, in addition to the cognitive recognition of what constitutes quality in software development.

The values identified above are professional in the broadest sense. CC2001 specifies that graduates should

Be guided by the social, professional, and ethical issues involved in the use of computer technology.

In commenting on the need to include Professional Practice in the Curriculum, the Joint Task Force says

Accreditation bodies, however, usually require not only that students *acquire* these skills—either through general education requirements or through courses required specifically for computer science—but also that students *apply* these skills in their later courses.

It commends the threshold and modal (average) standards of performance for computing graduates specified in the UK's Computing subject benchmark (QAAHE, 2000). At threshold level, graduating students will be able to

- Identify appropriate practices within a professional, legal, and ethical framework.
- Appreciate the need for continuing professional development.

Whereas the average graduating student will be able to

- Follow appropriate practices within a professional, legal, and ethical framework.
- Identify mechanisms for continuing professional development and life-long learning.

This assumes that differing degrees of commitment to professional practice and life-long learning can be discerned; in other words, it implies assessment in the affective domain.

CC2001 discusses how professional practice can be assessed and says

The assessment process should encourage students to employ good technical practice and high standards of integrity. It should discourage students from attempting to complete work without giving themselves enough time or in a haphazard manner, such as starting and barely completing work the night before an assignment is due.

but it gives no guidance on how to measure professional practice values and attitudes. This tension between what is in the curriculum and what we assess will continue to

increase in importance as the requirement to emphasize professionalization in the CS curriculum develops further.

Values and their assessment do not explicitly figure largely in the CS classroom. Efficiency and effectiveness of programming techniques are discussed, and implicitly recommended. In the few cases when researchers have explored whether values have been communicated, they have used attitude questionnaires as an instrument. Li & Prasad (2005), for example, investigated students' opinions on learning and accepting coding standards in programming courses, but this was to measure the effectiveness of the teaching the students had received, not their behaviour. Howles (2003) argued that CS educators should encourage the development of workplace skills "such as personal accountability, a strong work ethic, and an ability to deliver on-time and correct work". She attempted to measure student behaviour by a survey and proposed that student assignments be graded as unacceptable if they do not adopt appropriate quality controls (such as testing and compliance with deadlines.)

A considerably larger number of studies discuss how ethics can be introduced into the CS curriculum (Greening, Kay & Kummerfeld 2004, Werth 1997). Most propose the use of workplace-based vignettes or scenarios to trigger discussion of ethical issues. Students are assessed on their reflective responses to these presented examples, and are often expected to include reference to appropriate codes of practice and applicable legislation. None of them have seen fit to judge the students' expressed attitudes as worthy or unworthy, as right or wrong, or as professionally more or less appropriate.

#### 4 Assessment in the Affective Domain

Although, as shown above, CC2001 includes learning outcomes in the affective domain at the programme level, there are no examples of affective learning outcomes in its CS course descriptions. This section proposes some ways in which assessment in the affective domain could be incorporated, using the taxonomy outlined in section 2.

The importance of testing is a key aspect of software quality and students are introduced to it in every introductory software development course, whether Java programming or spreadsheets. A possible learning outcome in the affective domain for an introductory course could be that students will "appreciate the importance of testing and act upon it in developing software". The difficulty is that it is probably impossible to assess this at any level higher than Responding, because students will be heavily prompted to test during the delivery of the course in order to instil good practice. Even so, log books or a commentary on the development of the software could indicate whether students had responded reluctantly or from a commitment to testing.

An additional difficulty is that what is sought is long-term commitment, not a belief that evaporates as soon as the final assignment has been submitted. This could be addressed by putting the learning outcome in a later course involving software development, in which there is less explicit teaching of testing. It would then be possible

to detect commitment to testing through the approach students took to producing a software artefact, that is to assess whether the student is Responding or Valuing. In the context of a final-year project, an affective learning outcome could be that students are committed to quality and act accordingly in developing software systems. The project deliverables, including a reflective log and the supervisor's observations, could be used to assess this at the levels of Receiving, Responding, Valuing and Organization, because students could show that they had brought together aspects of quality management in their approach to their work. Level descriptors might be

**Receiving:** pays lip service to quality of software;

**Responding:** incorporates quality management in response to external prompting;

**Valuing:** uses previously taught quality management approaches effectively;

**Organization:** demonstrates effective planning and risk handling in a range of situations. Responds to project vicissitudes with aplomb.

This attempts to assess graduating students' degree of commitment to professional practice, as envisaged by the Computing Subject Benchmark and CC2001.

Commitment to life-long learning might appear impossible to assess before a student graduates, but an example taken from the authors' university suggests that this is not entirely the case. The context is a course in which students provide IT consultancy for local small and medium enterprises (SME). The nature of the work is such that students consistently have to tackle assignments that go beyond their existing technical knowledge. One of the learning outcomes of the course is "Students will be able to formulate and evaluate technical alternatives to meet IT requirements arising from small businesses" and one facet they are assessed on is technical knowledge and adaptability. The level descriptors used are

Meagre technical knowledge; makes little effort to acquire new knowledge required for assignments.

Limited technical knowledge; acquires new knowledge only to the bare extent necessary to carry out assignments.

Has or gains knowledge in depth relating to own assignments and is diligent in developing a good background of knowledge across the whole range frequently encountered in SME IT consultancy.

Has or gains knowledge in depth relating to own assignments and is assiduous in developing in-depth technical knowledge across the whole range frequently encountered in SME IT consultancy, and in keeping such knowledge up to date.

The students' performance against these descriptors is assessed through the assignment deliverables, a reflective report, feedback from clients and observation by the manager of the consultancy operation. The course convener reports that this appears to be measured consistently by a fairly large group of examiners. It provides a good indicator of the extent to which the

student has the necessary attitudes and values to satisfy the CC2001 requirement that CS graduates should “Keep abreast of current developments in the discipline to continue [their] own professional development”.

It is very difficult to realistically assess students’ attitudes to ethical issues that arise in industry because only their reported thoughts about hypothetical examples can be captured. Students’ lack of ethical behaviour can, of course, sometimes be observed directly, as when, for example, they cheat, plagiarize or misrepresent. We can also note ethical shortcomings when students show a lack of respect to fellow classmates or project team members.

## 5 Discussion

It is a fundamental tenet of Western society that people should be judged on their actual behaviour, not on their private beliefs. What people do is determined both by their cognitive knowledge and skills and by their beliefs, feelings and attitudes. Educators routinely assess across the cognitive domain and reward students for what they *can do*, but what they actually want is that the students *do do* what they have been taught. Behaviour depends on competence and the willingness to apply that competence (ten Cate & De Haes 2000), so competence is only a prerequisite for actual behaviour. If degree programmes exclusively assess competence they lead students to believe that willingness is less important.

There is a practical difficulty. Assessment in the affective domain is seldom explicit, with the result that there are no effective assessment instruments and few affective learning outcomes in curricula for instructors to draw on.

Is it sufficient that CS degree programmes limit themselves to developing and assessing competence? The answer must be “No”. Employers expect commitment to applying professional values, so sending out CS graduates who are unwilling to apply their competencies reduces students’ employability and harms the reputation of the institution they graduated from. This is reflected in the attributes of graduates set out in CC2001.

A key problem in developing CS students’ attitudes towards professional practice (in the widest sense) is a weak understanding of what learning experiences produce what changes in the affective domain (Krathwohl, Bloom & Masia 1964). The CS community sees internships and practice-based projects as particularly effective in developing a good work ethic and positive attitudes to professional practice, but there is no justification for not explicitly expecting professional behaviour across the whole of the CS curriculum. There is an ethical question about how far we should go. Use of pirated software to do coursework implies a disregard for professional principles; should it automatically lead to a fail grade?

Instructors face a dilemma: on the one hand they are concerned not to judge their students’ worth, but on the other they are constrained to ensure the moral sensibility of their professional graduates.

## 6 Questions for Discussion

Is it feasible to assess CS students’ attitudes and values as well as their cognitive skills?

Which are the main areas of CS in which we might improve learning and teaching by making learning outcomes in the affective domain explicit?

What assessment tasks could improve cognitive alignment with such affective learning outcomes? Can they easily be incorporated into the curriculum?

How do we link the requirement that the CS graduate should “follow appropriate practices within a professional, legal, and ethical framework” with institutional codes of integrity?

Is it ethical to assess CS students’ attitudes and values?  
Is it ethical not to assess students’ attitudes and values?

## 7 References

- Biggs, J.B. (1996). Enhancing teaching through constructive alignment, *Higher Education* **32**: 347-364.
- Bloom, B.S., Engelhart, M.D., Furst, E.J., Hill, W.H. & Krathwohl, D.R. (1956). *Taxonomy of Educational Objectives: Handbook 1 Cognitive Domain*, Longmans, Green and Co Ltd, London.
- Greening, T., Kay, J. & Kummerfeld, B. (2004). Integrating Ethical Content Into Computing Curricula. *Proceedings of the 6<sup>th</sup> conference on Australasian computing education*, 91.
- Howles, T. (2003). Fostering the Growth of a Software Quality Culture. *Inroads* **35** (2): 45-47.
- Joint Task Force on Computing Curricula (2001). *Computing Curricula 2001: Computer Science*, ACM.
- Krathwohl, D.R., Bloom, B.S. & Masia, B.B. (1964). *Taxonomy of educational objectives: the classification of educational goals. Handbook 2 Affective domain*, McKay, New York.
- Li, X. & Prasad, C. (2005). Effectively Teaching Coding Standards in Programming. *SIGITE’05 Proceedings of the 6th conference on Information technology education*, 239.
- Little, J.C., Granger, M.J., Boyle, R., Gerhardt-Powals, J., Impagliazzo, J., Janik, C., Kubilus, N.J., Lippert, S.K., McCracken, W.M., Paliwoda, G. & Soja, P. (1999). Integrating Professionalism and Workplace Issues into the Computing and IT Curriculum. *Inroads* **31** (4): 106-120.
- QA Agency for Higher Education (QAAHE) (2000). *Computing Subject Benchmark*. <http://www.qaa.ac.uk/academicinfrastructure/benchmark/honours/computing.asp>. Accessed 29 October 2007.
- ten Cate, Th. J. & De Haes, J.C.J.M. (2000). Summative assessment of medical students in the affective domain. *Medical Teacher* **22** (1): 40-43.
- Werth, L.H. (1997) Getting Started With Computer Ethics. *Proceedings of the 28<sup>th</sup> SIGCSE technical symposium on Computer science education*, 1.

# Puck - A Visual Programming System for Schools

Lutz Kohl

Friedrich Schiller University Jena,  
Didactics of Computer Science,  
Germany  
lutz.kohl@uni-jena.de

## Abstract

There are many visual programming languages for education but they are poorly used in German schools as most do not fit the requirements of the various curricula. This paper introduces a new visual programming language called Puck, that has been developed at the Friedrich Schiller University, Jena, in accordance with the wishes of teachers in Thuringia (a federal state of Germany). The visualisation in Puck is based on text-based programs, but syntax errors are prevented through the use of a visual system. Data types, variables, control structures, procedures and parameters can be taught with Puck. In addition, with code generation for Oberon-2, Java and pseudocode, students can be prepared for textual programming. Puck combines ideas from other visual programming languages and adds new features, such as the creation of correct expressions through context menus and a complexity calculation for a program.

**Keywords:** visual programming, puck, programming system, visualization

## 1 How the Visual Language Puck was created

In Germany, the education curriculum is separately defined by each of the 16 federal states. In some states computer science education is integrated with other subjects, in others it is a separate compulsory subject, and in still others it is an optional course. In addition, the contents vary between the federal states. While some topics, such as algorithms and e-mail communication, can be found in most curricula, other topics, such as logic programming and the software life cycle, are found in only a few.

Visual programming languages can be easy to learn, and they can prevent the students from making frustrating syntax errors, so why do most German schools still tend to use text-based languages for introducing programming? Firstly, the teachers have to learn the new visual language and it must fit in with the requirements of the curriculum. Next, there is the question of how easy it is to change from a visual to a text-based language. One additional problem seems to be that German schools are able to buy operating systems and office programs, but they seldom have money for programming systems. So for the most part free systems are used, and it is not economically feasible for a company to develop visual programming systems specifically for schools. Therefore most ideas in this field have been developed at universities. Unfortunately, such software is often supported for only a few years, and does not break into the market.

Copyright 2008, Australian Computer Society, Inc. This paper appeared at 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. Raymond Lister and Simon, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

In a first student project the visual programming languages AgentSheets, LabView and LogoBlocks were shown to five teachers. Afterwards the teachers were asked what requirements a visual programming language should fulfil to be suitable for use in schools in Thuringia. A set of requirements was developed from an analysis of these interviews. The teachers requested a system with visual bricks that could be combined to form a program, as in LogoBlocks. It should also be possible to teach variables, data types, control structures and procedures with parameters, thus satisfying the needs of the curriculum. Further requirements were the generation of Oberon-2 code, to prepare for textual programming, and the development of a tutorial and a homepage. In the author's diploma thesis (Kohl 2004) the visual programming language Puck was created. Afterwards this language was simultaneously tested in school and enhanced in further student projects in accordance with the teachers' wishes. Exercises and examples were created, and the system was exhibited at the CeBIT and Learntec fairs and presented at two German conferences about computer science in schools (INFOS 2005 and INFOS 2007). Puck is free and can be downloaded at <http://www.ipuck.de>.

## 2 What is Puck?

The Puck system is a visual programming language for beginners. It is related to imperative text-based programming languages, and concentrates on the prevention of syntax errors, which are described by Kelleher & Pausch (2005) as one of the largest and most frustrating challenges for novice programmers, particularly because most of the error messages caused by syntax errors are not easily understood by novices. A Puck program can always be executed at any stage during its creation. Bricks for instructions such as input, output, sound, assertion, draw line, if then, loops and procedure calls are combined into a visual program. A new feature in Puck is that expressions of different data types can also be made visual, and so syntax errors such as forgetting to close a bracket cannot occur. Variables of different types (Boolean and Integer) and procedures with parameters can be created visually. German pseudocode can be generated as well as code for Oberon-2 and Java. In addition, a complexity measurement method to compare different solutions has been implemented. Many concepts and algorithms that in the past could be taught only with text-based languages can now be taught, with the help of Puck, without any worrying about syntax.

## 3 Related Work

Making it easier to teach programming was always a goal in the history of visual programming languages. Two popular visual programming systems for introductory computer science are Alice and Scratch. The Scratch<sup>1</sup> system

<sup>1</sup><http://scratch.mit.edu> and Maloney et al. (2004)

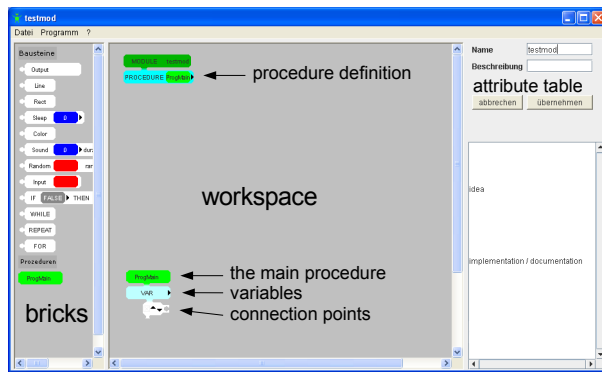


Figure 1: The Puck start screen

was developed at the same time as Puck and uses a similar approach to develop programs that are free of syntax errors. Scratch focuses on technological fluency at after-school centres with projects such as animated stories, games and interactive art. The Alice<sup>2</sup> system is also designed for the creation of animated stories, interactive games and 3D videos. Alice uses an object-oriented approach and can be used to introduce control structures, objects, classes, methods, attributes, inheritance, event handling and more. Alice is a powerful educational programming environment.

In contrast to Scratch and Alice, the Puck system has been developed explicitly for use in schools, in accordance with teachers' wishes, to prepare pupils to work with text-based programming systems. Typical first programming exercises that are used in imperative languages can be solved with Puck, and so switching to a text-based language should be easier. Compared with Scratch, Puck provides different data types and procedures with different kinds of parameters. In contrast to Alice, Puck is less complex and uses an imperative approach.

#### 4 How are Puck Programs created?

On the left side of the Puck programming window (Figure 1) are the bricks that can be selected by the students. These bricks can be dragged and dropped into the workspace in the centre of the window. When Puck starts up there are already a MODULE and a PROCEDURE in the workspace. The Procedure ProgMain has a white connection point that can be duplicated with its up and down arrows. The instruction bricks from the left side can be added to the connection points like pieces of a puzzle. Bodies of loops are visualized through indentation of connection points. Variables are created in the turquoise VAR field, and their names and types can be modified on the right side of the window in an attribute table. Integer variables and expressions are always blue, Boolean variables and expressions are always grey. The attribute table is also used to modify the instruction bricks. For example, after a click on the 'color brick' the user can choose whether the brick should set a new background color or a new foreground color in the attribute table. New procedures can be created with the arrow in the 'module brick'. After their creation, they appear under ProgMain on the left side. When a procedure has been added to the workspace, parameters and local variables can be created. Further instructions can then be added at the connection points. Procedures are called with a 'call brick' that automatically creates expressions for the actual parameters. Expressions in the Puck system are initialized to simple values (0 for Integer, FALSE for Boolean), which can be replaced by one of the locally known variables or parameters via a context menu. Expressions can also be expanded

<sup>2</sup><http://www.alice.org> and Cooper et al. (2003)

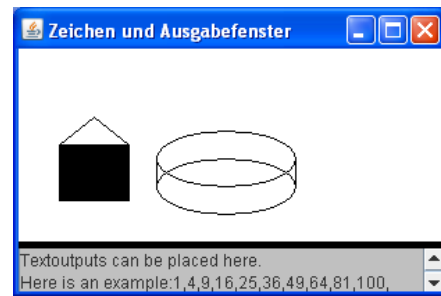


Figure 2: An executed Puck program

with the black arrow on the right. On every expansion, an operator and a new operand are added. If an operand is deleted, the next operation is deleted, too.

Only the operators that are possible for the type of the expression can be chosen. For Integer operators the user can choose between +, -, \*, DIV and MOD in a context menu. Brackets can be added around one operand via a context menu and they can be moved separately to the left or the right if this moving leads to another possible expression. This means that as each expression is created, the syntax is always correct. An example of an executed Puck program is shown in Figure 2. The window is divided into two parts. The text outputs appear in the grey area on the bottom, the graphic outputs in the upper area.

#### 5 Three Examples of Puck Programs

Many of the problems that are typically taught in the first programming lessons at school can be solved with Puck. One example is the tea timer program as shown in Figure 3. The program waits for an input to tell it how long the tea should brew in the cup. After this input the program counts down the remaining seconds. Then the program signals that the tea is ready with a sound and a green screen. This program is easy to understand, compact and not too mathematical, so it is well suited to beginners.

To stimulate the students' creativity a screensaver can also be programmed. Lots of options are possible with this exercise. A figure can fly over the screen, the background color can change slowly, or different circles can be shown on the screen.

The example program in Figure 4 teaches basic arithmetic. Two random numbers between 1 and 20 are chosen by the computer, and the sum, difference, and product of these two numbers are displayed. The user must now guess or calculate the original numbers and the computer checks the result.

As these three examples show, many problems can be solved with the help of Puck. However, limitations include that only the inbuilt bricks and the data types Integer and Boolean can be used. Furthermore Puck is not object oriented. More examples of Puck programs and publications can be found on <http://www.ipuck.de>.

#### 6 The Special Features of Puck

There are many features that were designed for school, as the visual programming language Puck was specifically developed to meet the requirements of teachers.

**No syntax errors.** In text-based programming courses, the teacher often has to spend time correcting syntax errors on the students' computers because beginners cannot deal with the confusing error messages. In Puck, they do not get any error messages because the bricks can only be connected in the correct manner. Syntax errors are not possible, and the program is executable at any stage of its development.

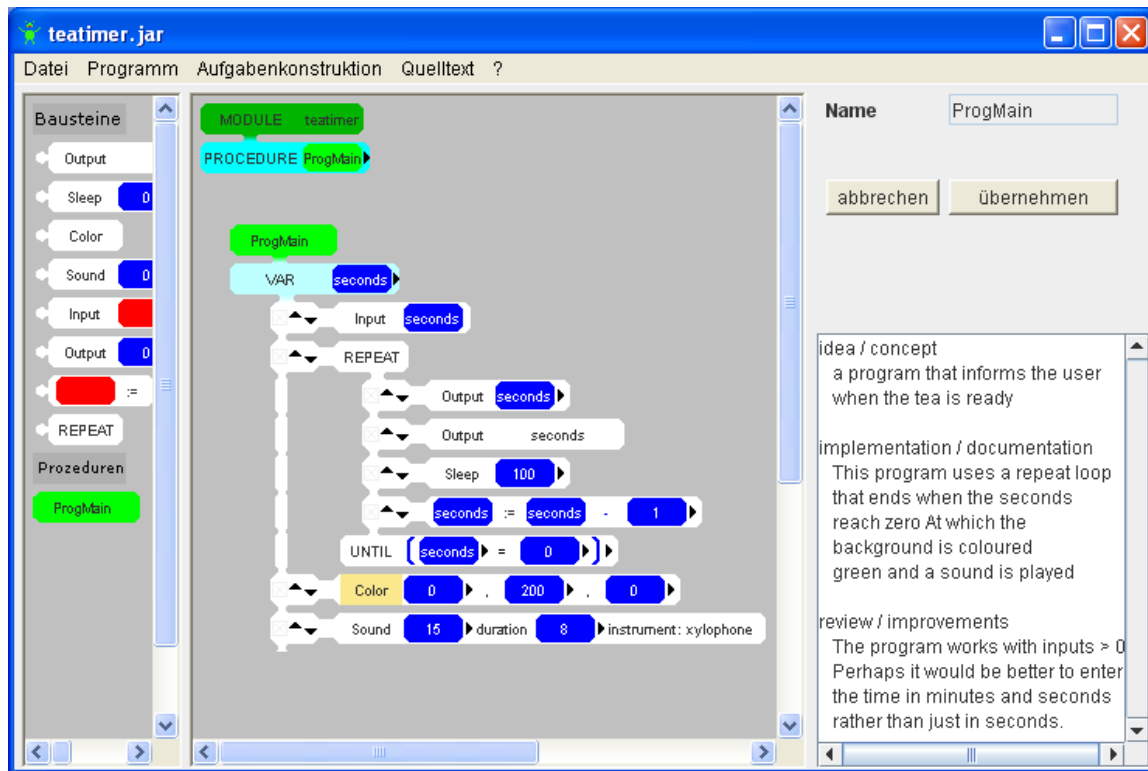


Figure 3: A tea timer program created with Puck

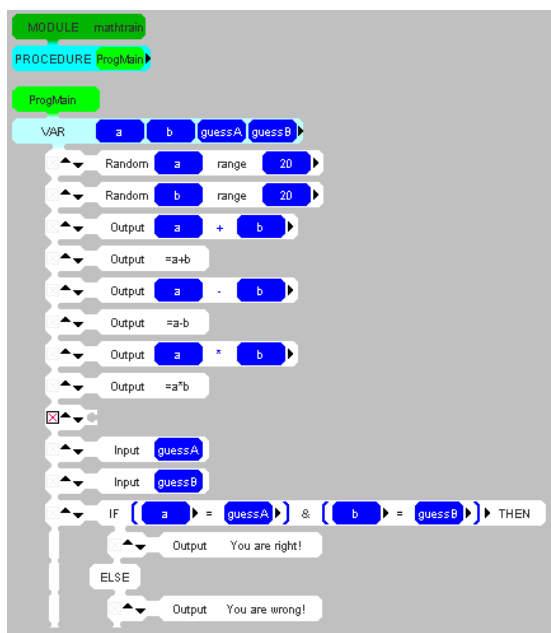


Figure 4: A maths trainer program created with Puck

**Motivating bricks.** Bricks such as input, output, color, rectangle/ellipse, line and sound have been developed, so that the users can see and hear the outcomes of their programs directly. Interesting exercises are possible with the 'sleep brick' and the 'random brick'. In combination, the collection of bricks can be quite motivating.

**Teacher can specify which bricks may be used.** Teachers may use the option screen to choose which bricks can be used by the pupils. For example, in the first lessons there might only be the 'line brick' and the 'color brick'. After five lessons, alternatives and loops might also be available, and after ten lessons, all bricks including procedure calls might be provided.

**Renaming variables affects the whole program.** If the user changes the name of a variable in the declaration, its name will change at all locations in the Puck program where this variable is used. This refactoring method is also used in professional IDEs, where it helps avoid syntax errors.

**A text field for comments is always visible.** In the visual programming window there is a small text field on the right. The students' ideas, concepts, comments, problems and reviews can be saved here, along with instructions by the teacher. The text stored in this field is always saved and loaded every time the Puck program is saved and loaded. This field for comments has the advantage that it is always visible for pupils.

**Complexity calculation.** Sometimes one problem may be solved in a number of ways. The teacher may say that one solution is more complex than another one. But what are the criteria for determining complexity? A complexity measurement method for programs, as defined in Rechenberg (1986), has been implemented in the Puck system. In this method, statement complexity<sup>3</sup>, data complexity<sup>4</sup> and expression complexity<sup>5</sup> are added to measure the overall complexity of a program. With this value the teacher can start a discussion about the complexity of different solutions. Perhaps teachers could also use this measurement to decide which programs can be taught in one hour, and which are too complicated.

**Code generation for Oberon-2 and Java.** At any time, code can be generated for the languages Oberon-2 and Java, so that students can create an initial (syntactically

<sup>3</sup>Every normal statement gets one point. Loops and alternatives get 3 points. If a statement is nested its points will be multiplied by 1.5 for every nesting level.

<sup>4</sup>The use of a predefined variable in a statement gets one point. The use of a predefined procedure in a statement gets two points. Every name is counted only once per statement.

<sup>5</sup>Operators used in expressions get points. <, ≤, ≥, =, ≠, +, - get one point. DIV, \*, NOT get two points. AND, OR, MOD get three points. The points of an operator are multiplied by 1.5 for every enclosing pair of brackets.

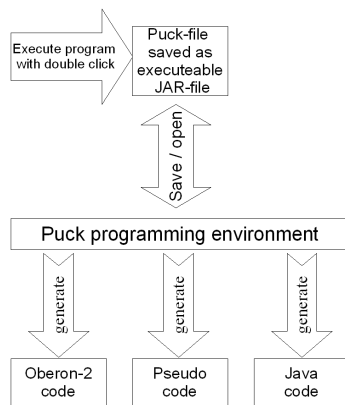


Figure 5: Different kinds of code generation in Puck

correct) program with Puck, and then continue their work on the text-based program. (Oberon-2 code generation was a requirement by the teachers because this language is used in the Thuringian curriculum in the 11th and 12th grade and in the exams. The functionality of some bricks is not implemented in the standard Oberon-2 system, so for these bricks the code cannot be generated.)

**Teacher mode with pseudocode generation, screenshots and example generation.** In teacher mode, German pseudocode can be generated to develop easy exercises where the students translate a given pseudocode program into a computer program. Different kinds of code generation are shown in Figure 5. In addition, teachers are able to generate screenshots and example programs – programs that can only be executed with a double click, and cannot be opened or modified with Puck.

**Log files.** To restore programs, and to analyze what the students have done, teachers can opt to have log files created. The activities of a day are stored in a csv-file, and every time a program is executed, saved or opened, a copy of this file is stored in the log folder. These copies can be opened with Puck, allowing the teacher to analyze exactly what the pupils did during the lesson.

**Help system.** The Puck help system illustrates the use of the whole system. All bricks are explained with simple examples. The functionality of the control structures, procedures and variables is described. In addition, a more complex example program, which calculates the days that have passed since a user's birthday, illustrates the use of all the functions together.

## 7 Evaluation

In 2005-2006, Puck was tested in four computer science courses in the 11th grade. The intention of this test was to improve the visual programming language. With the help of the teachers, bugs were corrected and new ideas such as Boolean input were added. In 2006-2007, Puck was tested in six schools, in the 8th or 10th grade, for two lessons every week for half a year or equivalent. The teachers were taught how to work with Puck in three half-day seminars at the Friedrich Schiller University, Jena. They were given materials with more than 50 exercises that can be programmed with Puck. The six teachers were pleased with the Puck system. Only one teacher had technical problems running Puck on the older computers at his school. All six teachers said that the removal of syntax errors in Puck was good for introducing programming. This simplification did not cause any other problems during the

evaluation. Five teachers had more time for the individual support of the pupils. In particular, some fast learners were able to work on their own. Five teachers agreed that after half a year with Puck, the pupils were prepared for the use of a text-based programming language.

The teachers did not utilise the special features of Puck very often. The help system was praised, and especially in the first lessons five of the teachers specified which bricks could be used by the pupils. Code generation, log files and complexity calculation were little used. Perhaps the teachers did not see an advantage for their work in these features. Furthermore, teachers must be competent in the use of a feature if they want to use it in their lessons. For example, a teacher who wants to use the complexity calculation should first know how this calculation works.

All in all, the teachers rated the Puck system as good. The fact that they will all use it again in the following year shows that they thought Puck was a good system to introduce programming.

## 8 Open Questions

A number of questions remain to be explored. Are special features such as code generation, complexity calculation, log files, a teacher mode and a comment field useful? Why did teachers use the special features so little? Do the pupils who used Puck perform better when they later work with a text-based system? What do we have to do to bring visual programming languages into schools?

## 9 Acknowledgments

The author would like to thank Susanne Fritsch, Mike Erler, Nico Günther, Mathias Gunkel, Alexander Lärz, Gabor Meißner, Friedrich Rau and Florian Rohde for their work on the Puck system and on the exercises. Thanks, too, to all the teachers who defined the requirements, tried Puck in their schools, and contributed helpful feedback.

## References

- Cooper, S., Dann, W. & Pausch, R. (2003), Teaching objects-first in introductory computer science, in 'SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education', ACM Press, pp. 191–195.
- Kelleher, C. & Pausch, R. (2005), 'Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers', *ACM Comput. Surv.* **37**(2), 83–137.
- Kohl, L. (2004), 'Entwurf und Implementierung einer visuellen Programmiersprache für den Einsatz in Schulen', Diploma thesis. Jena, Germany.
- Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B. & Resnick, M. (2004), 'Scratch: A sneak preview', *Second International Conference on Creating, Connecting and Collaborating through Computing* **00**, 104–109.
- Rechenberg, P. (1986), 'Ein Maß für die softwaretechnische Komplexität von Programmen', *Informatik Forschung und Entwicklung* **1**, 26–37.

# Effectiveness of Integrating Program Visualizations to a Programming Course

Essi Lahtinen

Tuukka Ahoniemi

Anniina Salo

Institute of Software Systems  
Tampere University of Technology,  
PO Box 553, Tampere, Finland  
{essi.lahtinen, tuukka.ahoniemi, anniina.salo}@tut.fi

## Abstract

To gain optimal benefits from program visualizations, their use in courses should be consistent. Instead of just covering all the topics they should also cover all the different learning situations and cognitive levels relating to the subject. We have integrated visualizations accordingly to our programming course and present some results about their usage in this paper. Our students showed a significant interest in using the visualization tool voluntarily.

**Keywords:** Program visualization, course integration

## 1 Introduction

As the concepts needed in programming are abstract and thus difficult to understand and apply for many students, it is tempting to use visualizations to concretize teaching programming. However, research on the field of using visualizations in programming education has shown very inconsistent results: on the one hand visualizations have been useful for students (Nevalainen and Sajaniemi, 2006; Ben-Bassat Levy et al., 2003; Ahoniemi and Lahtinen, 2006) and on the other hand they can even distract the students' attention from the essentials (Hundhausen, 2002). A large metastudy by Hundhausen et al. (2002) concludes that how the visualizations are used is more important than what is shown to the students.

The aim of our long-term course development has been to find how to use program visualizations most beneficially in an introductory programming course. We have developed ideas on the content and usage of visualizations, technical means for implementation, and a model on how to integrate the visualizations into the teaching. We have now combined these developments and our students are using visualizations as optional learning tool in an introductory programming course. This article analyses students' use of visualizations when guided this way, and their opinions on the visualizations.

## 2 Basis for Integrating Visualizations to the Programming Course

The compiler is an essential tool in a programming course. It is not used all the time, and definitely not for all purposes. For instance, when designing a program the student should not be using the compiler. Nonetheless, the students always know when

they need the compiler and find it at the right moment for its use. If we want to make the use of visualizations beneficial for the students, they should have a similar understanding of when to use the visualization tool. This is our aim in integrating the visualization tool to the course.

Naps et al. (2003) emphasize that user engagement is important in making the visualizations effective for learning. On this basis we have developed ideas on how to make students work with the visualization instead of just watching it. To make this kind of visualization technically possible we developed a mechanism called annotations for our visual interpreter tool. Further, both to improve the engagement and to make the visualizations as beneficial for students as possible, we developed a model for integrating the visualizations into the teaching. This work is summarized in the following subsections.

### 2.1 Supporting Higher Cognitive Skills with Visualizations

It seems that the biggest problem for novice programmers is not so much understanding basic programming concepts as learning how to apply them (Winslow, 1996). We have developed ideas on how to produce visualizations for novice programmers to enhance not only the students' understanding of concepts but their active programming skills. The ideas for developing visualizations are divided into different levels according to Bloom's taxonomy of the levels of cognitive development (Lahtinen and Ahoniemi, 2005). The categories are presented, with brief explanations, in Table 1.

The aim of developing this categorization of visualizations was to help visualization tool developers and teachers recognize the different kinds of skill that need to be supported in learning programming. Programming courses are not only about introducing programming concepts; hence there is no point in providing only *illustrative visualizations* for the students. Illustrative visualizations can be used when a new programming concept is introduced, but after that we need to support the students' learning process with the other types of visualization. This way the visualization tool becomes useful through the whole course, and integration of the visualizations and the course becomes more realistic.

### 2.2 Annotations for Defining Interactive Instructions

To be able to build visualizations on all the levels of cognitive development as described in the previous subsection, the visualization tool needs to interpret program code and give feedback on code written by the student. To enable this we developed a mechanism called annotations.

Level of the taxonomy	Visualization category	Characteristics and examples
1. Knowledge & 2. Comprehension	Illustrative visualizations	Presenting concepts, demonstrative, used to introduce a new subject
3. Application	Utilizing visualizations	Modify some parts of the program code to make certain changes to the functionality of the program, use visualization to better understand the program and the changes
4. Analysis	Problem-solving visualizations	Debug broken program code, analyse program behaviour, use visualization to help understanding
5. Synthesis	Productive visualizations	Use the visualization tool as a visual debugger when the student independently writes program code, or visualize the assignment description
6. Evaluation	Discerning visualizations	Visualizing solutions to a problem and comparing or evaluating them

Table 1: Bloom's taxonomy applied to visualization

A program visualization tool typically shows the visualized program code in a window that can include short comments as a part of the code. However, the purpose of the visualization is to thoroughly explain the program's execution to the student. To aid this, the visualization tool can include extra instructions in a separate window to describe the program and to draw the student's attention to the essentials.

To improve the quality of the instructions, the visualization tool developer can add annotations to define the visibility of the instructions. For example, a certain instruction could be shown only when the execution reaches the statement for the first time. When the execution returns to that statement there will be a new, more current instruction. This way the teacher can make the instructions more useful and interesting for the student. For example, when a function call statement is first encountered, the teacher would probably want to explain how parameter passing happens. When the function call finishes and the execution returns to the same line, the teacher can explain the passing of the function's return value.

In a similar way, taking the benefits of using an interpreter, it is possible to add some instructions that will be shown to the student depending on the state of the program, e.g. values of desired variables. The student can then be shown specific guiding instructions if his own program code does something wrong, or congratulated if the program state is correct. This idea is explained more in detail in an earlier article (Lahtinen and Ahoniemi, 2006).

With these special annotations that can access information about the program state, we have been able to build *utilizing* and *problem-solving visualizations* for the needs of a CS1 course. These appear to the student as normal interactive exercises that they can use when deepening their knowledge on a subject.

### 2.3 Integration of the Visualizations and the Course

Even though there are numerous visualization tools available for use in programming courses, the visualizations are still often not integrated in the learning situations. If there are a couple of visualizations available every now and then during the course, students do not develop the habit of using them regularly and they are easily forgotten. Besides covering all the subjects handled, the visualizations should cover multiple cognitive levels within one subject so that the student can benefit from them throughout the course. To gain maximal benefit from the visualizations, the teacher needs to plan their use precisely and actively guide the students in using them.

One flaw in the use of visualizations is that the

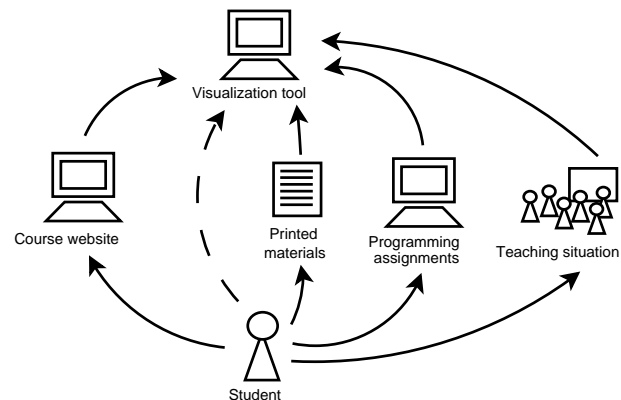


Figure 1: A model for integrating the visualizations into the rest of the course

students do not find them or do not realize the right time to use them. This occurs when the visualization tool is in a separate learning environment, e.g. on a web page that is not connected to the course material. Other little details, such as the need to type a password for the web page, may also hinder the use of the visualization tool.

To increase the use of the visualization tool it should be integrated with all the other course materials. We have used a model, (Lahtinen, 2006) illustrated in Figure 1, where the teacher includes links and references to the visualizations throughout the course material. The printed course material contains identification numbers for the visualizations, which are available on the front page of the visualization tool. The course web page where we publish the weekly exercise session materials contains hyperlinks to the related visualizations. Students can do some parts of the assignments using the visualization tool, and it is also used in some of the teaching situations. The idea of this model is to exploit the fact that students already know how to use the printed course materials in studying and they have to use the course web page to get all the practical information. With all the links and references, even the students who do not plan their learning situations and their use of the materials will benefit from the visualization tool when appropriate.

### 3 Target Course and Research Setting

The integration described in the previous section has been used in an introductory course for programming (CS1) in Tampere University of Technology. Prior studies examined the effects of integrating the visual-

Table 2: Subjects of pre-exercises in different weeks and the visualization categories they fall into

Ex.	Topic	Visualization type
2	If statements	Utilizing visualization
3	Loop structures	Problem-solving + utilizing visualization
4	Arrays	Problem-solving + utilizing visualization
5	Functions	Utilizing visualization

izations with the printed course materials (Lahtinen, 2006) and the use of visualizations in exercise sessions for a test group of students (Ahoniemi and Lahtinen, 2006). Now we wanted to explore whether and how the students would voluntarily use visualizations when doing small programming assignments.

In addition to the normal programming tools we made a visualization tool available to the students. The weekly exercise sessions of the course, which were not compulsory, included an optional pre-exercise assignment. Students who opted to do this pre-exercise could do it with pen and paper, with the help of an ordinary compiler, or with the visualization tool. The web page where the pre-exercise assignments were given had direct links to the same assignments in the visualization tool.

The visualization tool we use in our course is called VIP (Visual InterPreter) (Virtanen et al., 2005). It is based on an interpreter that uses a simple subset of C++ (namely C--). The interpreter accepts regular source code files, executes them, and automatically illustrates the program step by step.

Using the annotation technique described in Subsection 2.2, we prepared pre-exercise assignments for four exercise sessions during the course. Two of these were either to modify an existing structure or to add a clearly specified new functionality to a given program, and were thus *utilizing visualizations*. The other two exercises had two parts: a *problem-solving visualization*, i.e. a task to find out what the program does, and a *utilizing visualization*, i.e. to make changes to the functionality of the program. The subjects of the pre-exercises are presented in Table 2.

During these four weeks we monitored the number of pre-exercise assignment submissions and especially the number of students who voluntarily used the visualizations when doing the pre-exercise. After the four weeks we also asked the students, in a short survey, to reflect on the usefulness of visualizations with the pre-exercises.

#### 4 Visualization Usage

Since students could choose each week how they did the exercise, there were numerous different combinations of behaviour. We identified some groups of student whose occurrences are presented in Figure 2. A quarter of the students seemed to appreciate the visualization tool since they had done the exercises using it every time they turned in a solution. On the other hand, almost half of the students did not use the visualization tool for any of the solutions. There were two groups of students who changed their opinion about using the visualization tool: 7% who did the early exercises without the visualization tool but moved on to using visualizations, and 2% who used the visualization tool in the beginning but stopped using it. A further 7% alternated between using the tool and not using it, and 16% used other combinations.

The drop-out rate of our course is high, as is typical for programming courses. Often the first sign of failing the course is that the student stops attending

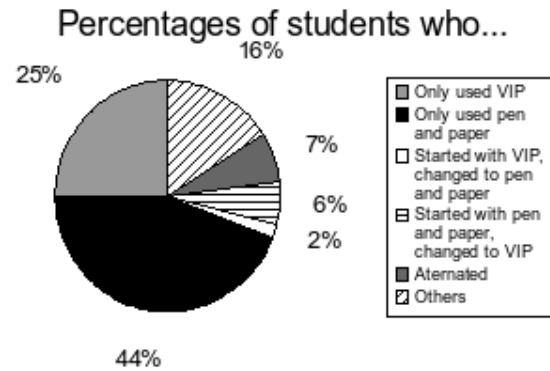


Figure 2: How students chose to do their exercises (n=308)

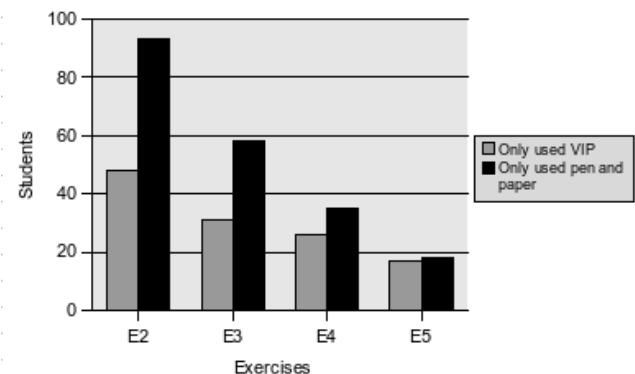


Figure 3: Numbers of exercises turned in by the students who always used VIP and those who always used pen and paper

the exercise sessions, or at least handing in solutions. Since there were two groups of students who had a clear opinion, either for or against using the visualization tool, we observed the drop-out rates of these two groups. Figure 3 shows the numbers of exercise submissions from students in these two groups. Activity decreased among students of both groups, but more sharply among the pen and paper group (a drop of about 80%) than among the visualization group (a drop of about 63%). It seems that students who consistently use the visualization tool are more likely to continue completing the assignments.

We asked the students who completed the assignments using the visualization tool to assess whether the tool was useful in the assignment. Figure 4 shows the students' opinions by assignment. The more difficult or abstract the content of the exercise, the better evaluation the visualization gets from students. For instance, students often perceive if-statements easy and functions more difficult.

There was no difference between the visualization types as presented in 2. The exercises where the visualization was seen as most useful were of different kinds: exercise 4 had both problem-solving and utilizing visualization and exercise 5 had only utilizing visualization.

#### 5 Discussion

In our earlier research we discerned that linking the printed course materials with the visualizations increases the students' use of the visualizations when they perform their course activities (Lahtinen, 2006). Integrating the exercise session assignments with the visualization tool was the next step towards full inte-

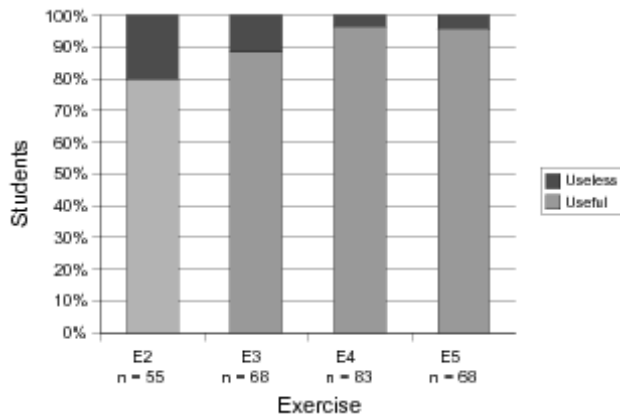


Figure 4: How useful students who used VIP found it in the assignments

gration of the visualization tool. Overall, the integration seems to be successful. A quarter of the students always used the visualization tool in their assignments even though its use was voluntary. About half of the students tried it at least once for completing an assignment.

The rates could be higher, but considering that using a new tool always requires some extra curiosity and effort, we know that some students will never do it unless it is obligatory. In fact, there is no need for all students to use visualizations. They can be a good aid for students with problems and completely useless for students who already understand programming well. We think it is best to let the students choose whether to use the visualization tool or not.

If we want to use the visualization tool more extensively, the next step would be to start using it to help with the bigger programming assignments, the only place where it is not now used. However, our goal is not to develop a programming course that uses visualizations for everything but to support the students with visualization where advantageous.

The students using the visualizations to solve the exercises appeared more likely to keep turning in the assignments weekly than the students who never used the visualizations. There was a clear difference in the drop-out rates of these two groups. The assignments can be a bit easier when you have the visualization to make them more concrete. However, we cannot conclude that it is the visualization tool that keeps the students doing the assignments. It might be that the most active students were also the most eager to use extra materials in their learning. The students in the other group were still active enough to do the voluntary pre-exercises, and their drop-out rate is still high. Whatever the explanation, visualizations seemed to be helpful since some students used them week after week. If the visualization can make the homework assignments more interesting for some students, by adding another perspective and thus keeping them working weekly, why would we not use it?

According to the collected data, the more difficult the topic, the more useful the visualizations become. This conflicts with the fact that visualizations are most often used only in the beginning of a course to present new concepts, and are forgotten later in the course. The visualizations on the higher levels of cognitive development and the integration model are a good start for developing the use of visualizations towards supporting the students in the more difficult topics and skills.

## 6 Conclusions

We guided our students so that using visualizations as extra material was easy and many of the students chose to use them voluntarily throughout the course. Another benefit we gained by using an interesting learning tool was that the students using it continued to take a more active part in the course. These results show that developing the use of visualizations in our course was worthwhile. We will keep improving their usage and their integration with the course.

## 7 Acknowledgments

The Nokia Foundation has contributed to the funding of this work.

## References

- Ahoniemi, T. and Lahtinen, E. (2006). Visualizations in Preparing for Programming Exercise Sessions, *in Proceedings of the Fourth Program Visualization Workshop*, Florence, Italy, pp. 54-59.
- Ben-Bassat Levy, R., Ben-Ari, M. and Uronen, P. A. (2003). The Jeliot 2000 program animation system, *Computers & Education* **40**(1), 1-15.
- Hundhausen, C. D. (2002). Integrating algorithm visualization technology into an undergraduate algorithms course: Ethnographic studies of a social constructivist approach, *Computers & Education* **39**(3), 237-260.
- Hundhausen, C. D., Douglas, S. A. and Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness, *Journal of Visual Languages & Computing* **13**(3), 259-290.
- Lahtinen, E. (2006). Integrating the Use Of Visualizations to Teaching Programming, *Proceedings of the conference Methods, Materials and Tools for Programming Education* pp. 7-13.
- Lahtinen, E. and Ahoniemi, T. (2005). Visualizations to Support Programming on Different Levels of Cognitive Development, *Proceedings of The Fifth Koli Calling Conference on Computer Science Education* pp. 87-94.
- Lahtinen, E. and Ahoniemi, T. (2006). Annotations for Defining Interactive Instructions to Interpreter-Based Program Visualization Tools, *in Proceedings of the Fourth Program Visualization Workshop*, Florence, Italy, pp. 34-38.
- Naps, T., Rössling, G., Almström, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velazquez-Iturbide, J. (2003). Exploring the role of visualization and engagement in computer science education, *SIGCSE Bulletin* **35**(2), 131-152.
- Nevalainen, S. and Sajaniemi, J. (2006). An experiment on short-term effects on animated versus static visualization of operations on program perception, *Proceedings of the 2006 int'l workshop on Computing education research ICER '06*.
- Virtanen, A. T., Lahtinen, E. and Järvinen, H.-M. (2005). VIP, a visual interpreter for learning introductory programming with C++, *Proceedings of The Fifth Koli Calling Conference on Computer Science Education* pp. 125-130.
- Winslow, L. E. (1996). Programming pedagogy – a psychological overview, *SIGCSE Bulletin* **28**(3).

# CLIP, a Command Line InterPreter for a subset of C++

Harri Luoma

Essi Lahtinen

Hannu-Matti Järvinen

Tampere University of Technology  
Institute of Software Systems,  
Tampere, Finland.

harri.luoma@tut.fi, essi.lahtinen@tut.fi, hannu-matti.jarvinen@tut.fi

## Abstract

C++ is not the best choice for a first programming language, but if it is used, the learning circumstances need to be as easy as possible. We have developed a pedagogically designed interpreter for this purpose. Our hypothesis is that an interpreter is easier than a compiler for a novice programmer to use. We do not want students to use language properties they do not yet fully understand, so our approach is imperative-first. In addition, when using an interpreter, learning concepts such as libraries can be postponed until later in the course.

C++ is a complex language and most of its language features are not needed by a novice programmer. Therefore we have simplified the language to a subset of C++ that we call C--. For instance, classes have been omitted. We have also put a lot of effort into producing clear, informative error messages, something that is made possible by of the simpler programming language.

This article introduces some other C/C++ interpreters, their evaluation, and the description of our interpreter called CLIP. So far CLIP has not been used by students, so its evaluation is left as future work.

**Keywords:** interpreter, C++, novice programmers, education

## 1 Introduction

To make the start of learning programming as easy as possible for our students, we have developed a tailored interpreter to address the needs of novice programmers. For external reasons, we have to use C++ as the programming language for our first programming course. However, to keep the concepts needed in the first programs as simple as possible, our teaching approach is imperative-first. Since C++ is such a complex programming language, we have chosen a subset of C++ to use in the beginning.

Our hypothesis is that using an interpreter is simpler than using a compiler. One of the reasons for this is that the feedback is received faster from an interpreter than from a compiler. When using a compiler, there is a lot of overhead in writing the first complete program. In C++, the student has to write *include* directives, *using* statements and a *main* function without knowing what they actually do. These are often experienced as mystical phrases or ‘spells’ since their

meaning is not yet understood. Using a compiler also requires the novice students to learn how to edit the program, save it, compile it, and, finally, run it. All these form a set of obstacles to the comprehension of the whole programming process. The use of an interpreter can help postpone the learning of these difficult phases.

In addition to eliminating the need for these ‘spells’, excluding the complicated language features, and simplifying the programming process, the purpose of our interpreter is to be able to give clear error messages in the students’ native language.

In this article, we introduce the CLIP C-- interpreter. We also present some reasons why the tool has been implemented and explain how it is going to be used. The article is organized as follows: In Section 2 we present the background of CLIP. Section 3 discusses novice programmer behaviour in general and Section 4 presents some other C++ interpreters for comparison. Section 5 describes CLIP and its programming language. Section 6 contains some evaluation, and Section 7 the conclusions.

## 2 Background

In teaching programming, the intention is to teach not just a programming language and its concepts but problem solving and programming knowledge in general. However, a real programming language is needed for practical training. Some teachers have tried a syntax-free approach to programming (Fincher 1999) where instead of a programming language, some kind of formal structures are used to present programs. This has not been well received, mainly because of the lack of motivation of students. They feel that they are not learning ‘real programming’ if no real-world programming language is used in teaching.

To address this problem, there are many languages designed especially for teaching novice programmers. Their approaches vary widely (Kelleher & Pausch 2005). One of the approaches to ease learning is to simplify an existing language, which normally leads to a separate language for teaching purposes.

Unfortunately, the teacher is often not free to base the choice of the first programming language on pedagogical criteria alone. For instance, the choice might be influenced by the university or the requirements of industry. Thus the teachers’ duty is often to make it as easy as possible for the students to learn the chosen language. Since in our context the pressure to use C++ is quite high, we have chosen to simplify C++ to get the benefits of a simple language.

## 3 Behaviour of novice programmers

Winslow (1996) has listed differences between expert and novice programmers, one of which is that the

novices do not have routines for the lower-level programming activities the same way the experts do. Thus, in the beginning it is essential to make the use of the programming tools as simple as possible, leaving the novice with more time and energy for learning the higher-level activities. This is the main reason why we decided to use an interpreter instead of a compiler.

Jadud (2006) has followed the edit-compile-cycle of novice programmers. He reports that students often get stuck with an error reported by the compiler and have problems in proceeding further. Surprisingly, a novice programmer can take hours to correct a simple syntax error that would require a couple of seconds from an experienced programmer. Perkins et al. (1989) report similar findings. They identified groups of novice programmers that behave in different ways when using a compiler: the stoppers, the movers, and the tinkers. The stoppers tend to give up when they face a problematic situation, while the movers try to explore the problem and often finally solve it. The tinkers can also be called the fast movers since their strategy is to try all different possibilities. Unfortunately, they choose the changes in a random way, so this kind of learning is not effective. Jadud's study also reveals that tinkering is a common strategy among novice programmers.

To prevent students from tinkering or stopping, and generally wasting their time with practical problems, the compiler's error messages should be made as informative and clear as possible. If the error messages directly give the first hint on how to proceed in correcting the program, the student can solve more problems without help, and hence get more confidence in his or her programming skills. This has been our goal when implementing CLIP.

#### 4 Existing C/C++ interpreters

Before implementing CLIP we familiarized ourselves with some other C/C++ interpreters. Kölling (1999) discusses object-oriented programming languages and environments and gives a set of requirements for them. Most of these requirements also apply to imperative programming environments: ease of use, integrated tools, support for code reuse, learning support, group support, and availability. We have used these requirements to evaluate the following interpreters.

**Ch (2007)** is a C/C++ interpreter that supports a superset of C with C++ classes. It extends the C language for scripting, numerical computing and 2D/3D plotting. It has a graphical interface called ChSciTE, where the code can be edited. We did not evaluate the commercial version, but found the free version unreliable.

**CINT (2007)** is a command line C/C++ interpreter aimed at processing C/C++ scripts. The language interpreted by CINT is a hybrid of C and C++. The syntax is less strict than C++, which means that not everything that runs on CINT can be compiled with a normal C++ compiler. There are also some differences in template libraries. Some C++ code may not work with it or may even be run in a wrong way. The interpreter is not very intuitive to use and is not meant for novice programmers or programming education.

**UnderC (2007)** is a fast C/C++ interpreter and is intended for programming education. It has a graphical editor, and it is working pretty well. There were some issues left in the GUI, and it

has been in beta phase since 2003, so we do not expect it to be developed further.

**IFNCP** (Sankupellay & Subramanian 2005) is an interpreter for novice C programmers. It is not available free of cost so we did not have an opportunity to test it. It is a web-based interpreter with an interactive online learning package but it only covers five basic C programming concepts. C++ is not supported.

Unfortunately, none of these interpreters meets our requirements or the requirements set by Kölling (1999). In brief, Ch is not reliable enough, CINT is not simple to use, and UnderC and IFNCP do not meet the availability requirement.

We also want to be able to add graphical features to our interpreter. It would be quite difficult to integrate a visualization module with these existing interpreters, so that is another reason for implementing our own interpreter.

#### 5 CLIP

The main ideas of CLIP are to support interpreter-based teaching and to give students better and clearer error messages in their native language. In addition to these, it should be possible to integrate a visualization module with the interpreter. This requirement has its roots in our visualization tool VIP (Virtanen et al. 2005), since we are not willing to maintain two separate interpreters.

C--, as we call the subset of C++ interpreted by CLIP, includes most of the basic structures of C++. We have selected the imperative paradigm for our elementary courses, which is a common approach when teaching C++ as the first programming language. Hence we have omitted some advanced features such as classes, namespaces, and exceptions. In addition, constructs that most often cause careless mistakes have been removed, or the interpreter deals with them more strictly.

This way, some of the problems of C++ can be diminished, if not avoided. We are aware that our changes do not make C-- an optimal language for teaching, but they do ease the task of the novice learner.

CLIP can be used in two modes, an interactive mode or a single-file mode. In the interactive mode, users write commands directly for the interpreter as they would be written inside the main function. The user can still define structs and functions normally. In the single-file mode, the user can give CLIP a single file, which is then interpreted and executed like any script. The command line interface of CLIP can be seen in Figure 1.

##### 5.1 The supported properties

The features of C-- were dictated mainly by the requirements of the first programming course where we intend to use CLIP. Some features were left out because they were considered harmful for students.

The main features that are missing are following:

- *goto* statement
- classes (except *cin*, *cout*, *string*, *vector*)
- file streams and string streams
- bit operators (to prevent confusion between *&* and *&&*)
- exceptions
- namespaces

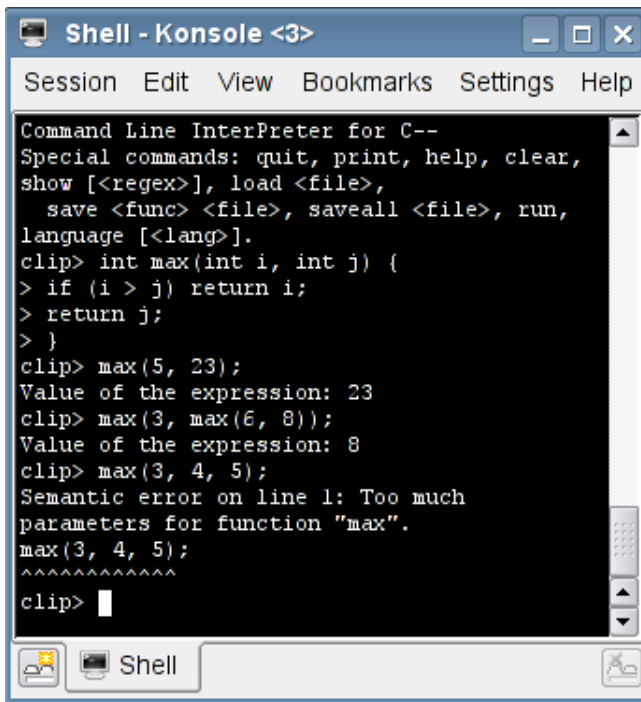


Figure 1: CLIP user interface

- templates (except vector)
- overloading of function names.

CLIP adds these features compared to a standard compiler:

- When an error is found from the code, the line that caused the error is shown.
- Error messages are localizable to student's native language.
- Contents of the symbol table are shown in runtime.
- We introduce *constraints*, which mean that the teacher can turn some features on or off depending on the topic to be taught.
- There are commands to save and load functions to and from external files.

## 5.2 Errors, warnings and other messages

There are eight different kinds of message: syntax, lexical, semantic, runtime, other errors, warnings, info messages, and constraints.

When an error occurs, it is shown instantly to the user, with no attempt to read the source code further. Further reading is usually needless because the later errors are often caused by the first error and do not clarify the situation at all. In fact, they may confuse the student and lengthen the error message output in vain. This approach has been proved to be viable in BlueJ (Kölling et al. 1999).

The constraints are a way to disable some C++ features from the interpreter. When disabled, the use of a feature will only produce an error message. At preset the following constraints can be enabled:

- switch statement
- pointers
- altering the for loop variable inside the for loop

- subroutines that have both non-const parameters and a return value (This constraint forces students to make a clear choice between a function and a procedure.)
- global variables
- arrays and vectors as members of a struct (This makes students use arrays of structs instead of a single struct with arrays in it.)

With the constraints, teacher can, for example, disable pointers when references are taught so that students cannot use pointers by mistake.

The interpreter also supports the following runtime error messages:

- divide by zero
- too deep recursion
- infinite loop
- reference through a null pointer
- index or pointer out of bounds
- use of uninitialized variables.

The output of CLIP is automatically split to fit the lines of the console. Individual words will not be split over multiple lines unless it is unavoidable.

## 5.3 CLIP compared to a normal compiler

The error messages given by a regular compiler, such as g++, are sometimes next to impossible to understand. This is often a consequence of the use of template libraries, but a missing semicolon in a vulnerable place can also cause very confusing error messages. The language supported by CLIP is much more constricted than C++, which makes it easier to check the syntax and to form clearer error messages.

In the next example we compare CLIP and the GNU C++ compiler (g++). We gave this piece of code to both of them:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> i;
    cout << i;
    return EXIT_SUCCESS;
}
```

The g++ compiler gave an error that was about 80 lines long and started with the following lines:

```
example.cc: In function 'int main()':
example.cc:6: error: no match for 'operator<<' in
'std::cout << i'
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/
../../../../include/c++/4.1.1/bits/ostream.tcc:67:
note: candidates are: std::basic_ostream<_CharT,
_Traits>& std::basic_ostream<_CharT, _Traits>::
operator<<(std::basic_ostream<_CharT, _Traits>& (*)
(std::basic_ostream<_CharT, _Traits>&))
[with _CharT = char, _Traits = std::char_traits<char>]
```

CLIP gave the following error

```
Semantic error on line 6: Can't print type
"vector of int" to cout.
    cout << i;
    ~
```

Clearly the error message of CLIP is easier for a novice programmer to understand.

In addition, when using CLIP, students would only need to write the following two lines to test the example code:

```
vector<int> i;
cout << i;
```

No struggling with namespaces or the main function is needed.

## 6 Evaluation of the tool

Since the interpreters discussed in Section 4 did not meet our requirements, we had to write an interpreter of our own. Hence, it is not a surprise that CLIP clearly meets our requirements. Unfortunately, we have not yet been able to gather student feedback on it. Tentative use by some faculty members has given encouraging feedback, so we are quite optimistic.

As mentioned above, Kölling (1999) lists properties of programming environments for the object-oriented approach. Of the applicable properties for imperative programming, CLIP does not provide group support at all, since in the first programming course each student should learn all the basic features individually. We believe that the rest of the properties have been met at least at the basic level, but without actual student feedback they cannot be evaluated.

However, we want to point out that in our opinion, CLIP gives good learning support. The main properties to support this have already been given: clear and informative error messages and the fast response of an interpreter. Furthermore, CLIP logs all the code the students input, so we get some data to examine. This data is intended to be used for improving the tool and the lectures. It will not be used for assessing the students.

Although we already think that CLIP is useful for our current course, utilizing its full potential requires rewriting the lecture material. The basic idea of the new material is that there will be no segment of program code in the material if the student does not understand why. We expect to have the first version of the new lecture material available for the academic year 2008-2009.

## 7 Conclusions

Selection of the first programming language for teaching should be done mostly on pedagogical basis. However, when facing cruel reality, this cannot always be the case. Our compromise has been to define C--, a limited subset of C++, and to offer a friendly interpreter, CLIP.

In this stage it is hard to tell how successful CLIP will be. With the new lecture material and integration of a visualization module, we expect to get a system in which the learner only uses the properties he or she knows, and in which the visualization will help the students not only to understand the program behaviour but also to debug their code.

## 8 Acknowledgements

The Federation of Finnish Technology Industries 100th anniversary and Nokia Foundation have partly funded this work.

## References

- Ch (2007). A C/C++ interpreter, <http://www.softintegration.com/> (referenced 18 October 2007).
- CINT (2007). A C/C++ interpreter, <http://root.cern.ch/twiki/bin/view/ROOT/CINT> (referenced 18 October 2007).
- Fincher, S. (1999). What are we doing when we teach programming?, in 'Proc. of the 29th ASEE/IEEE Frontiers in Education Conference', pp. 12a4-1-12a4-5.
- Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour, in 'ICER '06: Proceedings of the 2006 international workshop on Computing education research', ACM, New York, NY, USA, pp. 73-84.
- Kelleher, C. & Pausch, R. (2005). 'Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers', *ACM Comput. Surv.* **37**(2), 83-137.
- Kölling, M. (1999). 'The problem of teaching object-oriented programming, part ii: Environments', *Journal of Object-Oriented Programming* **11**(9), 6-12.
- Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (1999). 'The BlueJ system and its pedagogy', *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology* **11**(4), 249-268.
- Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989). Conditions of learning in novice programmers, in 'Elliot Soloway and James C. Spohrer, Studying the Novice Programmer', pp. 261-279.
- Sankupellay, M. & Subramanian, P. (2005). 'Teaching C programming with the aid of an interpreter - online interpreter for novice C programmer (IfNCP)', *Jurnal Teknologi* **11**, 33-44.
- UnderC (2007). A C/C++ interpreter, <http://home.mweb.co.za/sd/sdonovan/underc.html> (referenced 18 October 2007).
- Virtanen, A. T., Lahtinen, E. & Järvinen, H.-M. (2005). 'VIP, a visual interpreter for learning introductory programming with C++', *Proceedings of The Fifth Koli Calling Conference on Computer Science Education* pp. 125-130.
- Winslow, L. E. (1996). 'Programming pedagogy - a psychological overview', *SIGCSE Bulletin* **28**(3).

# Conflictive animations as engaging learning tools

Andrés Moreno

Erkki Sutinen

Roman Bednarik

Niko Myller

Department of Computer Science and Statistics  
University of Joensuu,  
PO Box 111, FI-80101 Joensuu, Finland  
`FirstName.LastName@cs.joensuu.fi`

## Abstract

In this paper we introduce the concept of conflictive animations and discuss its applicability in programming and algorithm design courses. Conflictive animations are animations that deviate from the expected behaviour of the program or algorithm they are supposed to present. With respect to the engagement taxonomy, we propose several ways of learning with conflictive animations. We also initiate a discussion about their possible benefits and drawbacks.

**Keywords:** Conflictive Animation, Algorithm Animation, Program Visualization, Engagement Taxonomy

## 1 Introduction

Many teachers have seen how *inadvertently* or *intentionally* being wrong in their lectures has sparked students' attention. In fact, some students will provide feedback only when they can correct the teacher. Moreover, when the teacher introduces errors intentionally, she can assess students' applied knowledge, i.e., whether the students can relate previous information to what is being explained.

Students in a classroom setting may be sufficiently distracted to not absorb what the teacher is saying. This would make it harder for them to later detect teacher's slips when they happen. Algorithm animation tools require the student to be more engaged, and allow for repetition of animations at different speeds according to students' abilities.

Asking programming students to find errors is not new. Rudder et al. (2007) created an online programming course whose materials included a *spot the error* task. Students were asked to identify the logical and syntactical errors in fragments of code, the goal being to evaluate students' comprehension of the basic programming rules.

A different approach is taken in MatrixPro (Karavirta et al. 2004) that addresses the dynamic process of algorithms. MatrixPro is an algorithm simulation tool that contains, amongst others, a *faulty exercise*. In this exercise the student is asked to reproduce the steps that will result in a "inconsistent search tree" according to an incorrect algorithm.

As far as we know, MatrixPro is the only algorithm animation tools that makes use of intentional errors, an incorrect algorithm. However, the possibilities of

intentional errors are not fully developed nor considered. MatrixPro designers regard the faulty exercise as a *completely open question* in their taxonomy for algorithm simulation exercises (Korhonen & Malmi 2004).

In this paper we first briefly explain algorithm animation, so as to later explain the concept of *conflictive animation*, and discuss its possible educational benefits. Section 4 suggests possible experimental studies to evaluate the appropriateness of conflictive animations in CS education. We finish the paper with a set of questions that we think are worth considering and the feedback from a survey.

## 2 Algorithm Animation

Algorithm animation tools visualize the execution of computer algorithms using graphical means. Usually in a step by step manner, animations show how algorithms modify and process given data. Data used in animations consist of characters, numbers, or more complicated data structures such as binary trees or queues.

Teachers and students have used animation tools to teach and study algorithms and programming. For example, sorting algorithms have been taught extensively using animations since Baecker (1981) produced the *Sorting Out Sorting* movie. Visualizations have represented data structures (Karavirta et al. 2004) and virtual machines (Moreno et al. 2004) using a set of graphical primitives: boxes hold values, bars indicate the *weight* of values, arrows serve as pointers, and graphs represent trees or queues.

Hundhausen et al. (2002) summarized the results of several studies on the effectiveness of algorithm visualization. Their meta-study revealed that algorithm animation is educationally effective to a certain extent, in part because of increased student motivation and the time spent with the animations. They emphasized the importance of the way students engage with animations, suggesting that students should regularly work with the animations, rather than just having animations as a uncommon aid.

### 2.1 Engagement Levels

Recent research (Naps et al. 2002) emphasizes the importance of engagement during interaction with algorithm animation. Naps et al. (2002) explicated several categories of engagement that animation tool creators should consider when designing algorithm animation tools. Their taxonomy lists six levels of engagement, as shown in Table 1.

This taxonomy can be used to help tool designers to add interactive elements to their designs as well as to test hypotheses about the effectiveness of a visualization tool. A way to engage a student with

Level	Well-behaved Animation	Conflictive Animation
<i>No viewing</i>	Tools make no use of animation.	Tools make no use of animation.
<i>Viewing</i>	Students can visualize animations, either step by step or continuously.	Students should be aware of the possibility of viewing an incorrect animation.
<i>Responding</i>	Students are asked to respond to questions related to an animation and the concepts presented in it.	Students have to detect the errors in the animation.
<i>Changing</i>	Students change the animation to explain a different concept that the animation was meant for.	Students change and correct a conflictive animation.
<i>Constructing</i>	Students use the animation tools to create an animation that explains an algorithm or a simpler concept.	Students purposely create a conflictive animation.
<i>Presenting</i>	Students verbally present an animation to an audience.	Students present the conflictive animation and try to confuse peers.

Table 1: Engagement levels and conflictive animations

a visualization is to ask the student to respond to the question ‘what happens next in the animation?’ (i.e., the program/algorithm execution) (Naps 2005, Myller 2007), or to construct an algorithm visualization using graphical primitives (Karavirta et al. 2004, Rößling & Freisleben 2002).

A complementary way to encourage students to respond to questions during animation would be to ask them to spot errors in it. In this way the question is asked before the animation and they need to pay attention to the animation, a) to understand it and b) to know when an error occurs.

### 3 Conflictive Animation

Constructivism tenets claim that a student creates knowledge by combining previous knowledge with recent experiences. The resulting mental model may hold misconceptions and not be *viable*. Learning, then, will only happen when students’ mental models adapt to the new information and make students reject misconceptions (Smith et al. 1993-1994). The teacher’s task is to make students’ misconceptions explicit and to guide students towards a viable mental model. Algorithm animations present an opportunity to help teachers and students to identify misconceptions. Conflictive animations are designed to be a tool used by students to reflect on their own mental models. Students should view animations critically, looking for possible errors in them.

We define conflictive animations as those that deliberately do not reproduce what the animated code or algorithm is programmed to perform. An animation would advance normally until it starts showing an alternative and potentially wrong execution path. The animation may eventually come back to normality, e.g., showing the correct output. Students should identify when the animation goes wrong, stop it, and reflect on the detected error. The possible errors can affect several levels of the animation, from a simple comparison gone wrong, e.g., evaluating  $3 > 4$  as *true*, to higher-level algorithmic concepts, e.g., skipping a balancing step in a red-black tree. An extreme case would be to show an animation of a quicksort algorithm when explaining heapsort.

The idea behind conflictive animations is to add a new dimension to the engagement taxonomy. Conflictive animations could be part of every level in the taxonomy, as shown in Table 1, complementing the educational value of working just with *well-behaved animations*. Students usually have a mental model of how a specific program works before they actually visualize it. Thus, visualizing it should act as a mere proof-check of their beliefs. However, students’ knowledge is frequently *fragile* (Oliver & Dal-

bey 1994), meaning it is incomplete, not recalled, or maybe just wrong. Thus, this would mean that students should benefit from the correct animations: they are complete and correct.

Conflictive animations are graphically described in the Venn diagram shown in Fig. 1. Circle A represents steps in the conflictive animation for an algorithm, and circle B represents steps in which the student spots an error. The intersection of the circles represents the situation in which the student correctly spots the error, a *true positive*. The rest of circle B represents *false positives*, in which the student mistakenly detects an error while the animation is behaving well, and the rest of circle A represents *false negatives*, in which the student fails to detect an erroneous step.

Conflictive animations should try to reveal one or two possible misconceptions that students might have. But students should not only check for the correctness of the visualization. Meaningful conflictive animations should require students to question their own mental models while the animation is running. Students’ false positives could reveal further misconceptions beyond those being examined by the animation.

Unfortunately, students commonly use the visualization as just another way to have an algorithm or program executed, focusing only on the final result (Moreno & Joy 2007). This eliminates the potential of animation tools to help students create a viable mental model of the algorithm execution. Applications try to overcome this by increasing the students’ engagement level. Some tools ask students to predict the next step in an algorithm, or even to create an animation for themselves.

At times, the answers to the questions posed by the tools can be guessed from cues taken from the last image shown, or from the algorithm pseudo-code. Thus they do not always engage the students in a meaningful learning activity. Furthermore, asking students to produce a new animation may require advanced skills (Rößling & Freisleben 2002) that students would struggle to acquire.

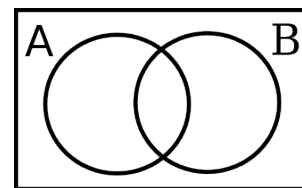


Figure 1: Venn Diagram for Animations

Conflictive animations would require students to follow them carefully step by step. Students should match the graphical representation with the code or concept it actually represents. To be able to spot errors in the animation, students would first have to understand the conventions of the animation, i.e., the meaning of the graphical metaphor it uses to represent the execution. Thus we believe that finding where animations have gone wrong would not only improve students' learning on a particular concept, but give them a better basis for understanding subsequent animations.

### 3.1 Scenarios

Students should be acquainted with the concepts before testing their knowledge and comprehension with conflictive animations. Their teacher should have already explained the concepts, probably with the use of animations. Then students could view the conflictive animations, knowing that they are conflictive, and try to spot the errors.

Three major roles can be identified in the common scenarios: students, teachers, and peer students. Students could use the conflictive animations to evaluate themselves, or teachers could use them to formally assess students. Peer students could build conflictive animations to challenge other students.

In the first case, the tools should give students instant feedback once they believe they have found the error in the animation. Feedback should at least indicate the correctness of their answer, and a brief explanation of what went wrong in the animation.

The second case could involve written assignments in which students would describe in detail what went wrong in the animation and when. Teachers would be able to detect the common misunderstandings and give personalized feedback to the students.

Finally, conflictive animations could be the basis for a gaming environment. Some students could design and build a conflictive animation, and the other students would try to find the errors in the animation their peers have designed. While building any animation is a complex task, the competitiveness of the assignment may provide additional motivation to students.

### 3.2 Animations as a story

Interest in CS has seen a drop and student intakes across different CS departments have sharply reduced (Vegso 2005). Alice (Moskal et al. 2004) is an animation tool meant for programming movies and games in an accessible way. Moskal et al. (2004) have shown that Alice appeals to a wide audience and its use increases students' performance and their interest in computer science. Alice's best asset is the ease with which people can use it to *implement* a movie or a game. But another important asset is the playful environment it provides. Students can add a variety of everyday characters and objects and program them to act according to the students' own ideas.

We suggest that algorithm animation tools should add animated characters to animations in order to retain students' interest in the topic, and to attract new students to major in CS. Characters could guide, or misguide, the animation blocks, and describe the animation behaviour. This way, animations will not only consist of boxes, lines and dots moving around, as in *Sorting Out Sorting* (Baecker 1981), but they will represent a story that can attract a diverse audience.

We envisage an animation story in which two characters collaborate to implement an algorithm. If it

is a sorting algorithm, they could move boxes that contain the data to be sorted. However, one character knows the correct steps of the algorithm and the other is wrong. They will discuss and reason about the changes made, e.g., why an array element was selected to be the pivot element. Each will try to convince the other about how to perform the next step. Eventually, the *conflictive character* will manage to convince the *smart character*. This should prompt a reaction in the viewer, who wants the algorithm animation to finish correctly. A set of animations following this idea could be realized to create a multi-episode show to be used in CS lessons at high schools.

### 3.3 Implementation

Algorithm animation tools such as ANIMAL (Rößling & Freisleben 2002) provide the tools for *visualizers* to create animations of algorithms. We propose that visualizers consider the possibility of creating conflictive animations and uploading them to online repositories<sup>1</sup>. These conflictive animations should be clearly labeled as such to avoid misunderstandings. A start could be made by modifying the scripts of existing animations to introduce errors in them.

JHAVÉ (Naps 2005) could also be extended to support this new way of interaction. Students should be able to push a button to signal that the animation has gone wrong. JHAVÉ would then ask the student at what point the animation went wrong, perhaps using a graphical multiple-choice question that includes snapshots of the animation.

Teachers like using their own examples in their lectures, but they often complain about the time-consuming task of creating animations (Naps et al. 2003). Thus automatic generation of conflictive animations is also important. Teachers could use examples they are comfortable with, and students could test their knowledge with their own programs.

To cater to teachers' needs, generation tools should let the teacher modify certain parameters, e.g., the algorithm to animate, the initial data, the concept to assess, or the level of divergence from correct behaviour. JHAVÉ, for example, already implements the first two parameters. With this information, tools should be able to create an animation plan. Incorrect behaviour can be implemented in the inner algorithms that drive the animations. For example, two different visual routines for animating the balancing of a tree could be implemented, one representing the correct behaviour and the other the conflictive one. The latter would be triggered when the right conditions are met, e.g., a left insertion that prompts a rotation.

Completely automated generation of conflictive programming animations might show to be more difficult to accomplish, as programs created by the students follow a non-deterministic execution behaviour. Algorithms should produce meaningful conflictive animations from students' programs. They should also infer what concept should be conflictive, and when to animate it. A solution for this issue is adaptation, i.e., keeping a user model and a user goal map. Adaptation could keep track of users' current knowledge and learning goals, which vary as the course progresses.

## 4 Suggested Evaluation

Implementing the concept of conflictive animations is not a trivial task. Existing tools have to be redesigned, both the GUI and the animations they produce. In addition, the new learning task of spotting the errors in animations could confuse the students.

<sup>1</sup><http://www.animal.ahrgr.de/animations.php3>

Students should know what the task consists of – detecting an error – and they must be able to solve that task with the tool using the new GUI and animations. Therefore usability evaluation should be carried out from the very beginning in the design phase. Once the task is well defined and the tool is usable, it is time to evaluate the usefulness of conflictive animations.

We propose two main hypotheses to test the possible benefits of conflictive animations:

*H1:* We hypothesize that students using conflictive animations will understand the concepts explained better than students using a tool at the responding level in the engagement taxonomy, e.g., JHAVÉ. By revealing possible misconceptions, students' knowledge, once fragile, will crystallize. Students will check their assumptions with the animation and discard those that do not correspond to the correct animation.

*H2:* Students' motivation to view visualizations and animations will increase when they know there are potential mistakes in them.

## 5 Discussion

We have presented in this paper the concept of conflictive animations, which should reinforce correct mental models in students. The educative effect of intentional errors in instruction is not well researched, and could produce an adverse result. According to our beliefs, conflictive animations should at least increase the attention students devote to animations and their critical thinking.

From our point of view, conflictive animations can at least raise few interesting questions apart from the hypotheses mentioned above:

- Not everybody enjoys being deceived, but does everyone benefit from the deception?
- How can trust in a learning tool be maintained when it is constantly going wrong?

Based on the small-scale survey we carried out at the Koli Calling 2008 conference, CS educators have used some form of conflictive animations or examples and they would be willing to use a tool that provides conflictive animations. We also asked educators to order different scenarios based on their perceived effectivity on learning. We had also classified those scenarios based on the engagement taxonomy classification and interestingly the order of the conflictive animations perceived effectivity does not match the proposed order of the well-behaved animations (Naps et al. 2002). Based on the educators answers, the order of the engagement levels for conflictive animations based on their educational effectiveness from the most effective to the least effective is as follows: changing/correcting, responding/detecting, viewing, constructing, no viewing, and presenting.

## References

- Baecker, R. (1981), 'Sorting out sorting, color/sound film'.
- Hundhausen, C. D., Douglas, S. A. & Stasko, J. T. (2002), 'A meta-study of algorithm visualization effectiveness', *Journal of Visual Languages and Computing* **13**(3), 259–290.
- Karavirta, V., Korhonen, A., Malmi, L. & Stalnacke, K. (2004), MatrixPro - a tool for demonstrating data structures and algorithms ex tempore, in 'Proceedings of ICAIT 2004', pp. 892–893.
- Korhonen, A. & Malmi, L. (2004), Taxonomy of visual algorithm simulation exercises, in 'Proceedings of the Third Program Visualization Workshop', The University of Warwick, UK, pp. 118–125.
- Moreno, A. & Joy, M. (2007), 'Jeliot 3 in a demanding educational setting', *Electronic Notes Theoretical Computer Science* **178**, 51–59.
- Moreno, A., Myller, N., Sutinen, E. & Ben-Ari, M. (2004), Visualizing program with Jeliot 3, in 'Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004', Gallipoli (Lecce), Italy, pp. 373–380.
- Moskal, B., Lurie, D. & Cooper, S. (2004), Evaluating the effectiveness of a new instructional approach, in 'SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education', ACM Press, New York, NY, USA, pp. 75–79.
- Myller, N. (2007), 'Automatic generation of prediction questions during program visualization', *Electronic Notes Theoretical Computer Science* **178**, 43–49.
- Naps, T., Cooper, S., Koldehofe, B., Leska, C., Röbling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., Anderson, J., Fleischer, R., Kuittinen, M. & McNally, M. (2003), Evaluating the educational impact of visualization, in 'ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 124–136.
- Naps, T. L. (2005), 'JHAVÉ – addressing the need to support algorithm visualization with tools for active engagement', *IEEE Computer Graphics and Applications* **25**(5), 49–55.
- Naps, T. L., Röbling, G., Almström, 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 'ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 131–152.
- Oliver, S. R. & Dalbey, J. (1994), 'A software development process laboratory for CS1 and CS2', *SIGCSE Bull.* **26**(1), 169–173.
- Röbling, G. & Freisleben, B. (2002), 'ANIMAL: A system for supporting multiple roles in algorithm animation', *Journal of Visual Languages and Computing* **13**(3), 341–354.
- Rudder, A., Bernard, M. & Mohammed, S. (2007), Teaching programming using visualization, in 'Proceedings of the Web Based Education 2007 Conference (WBE)', ACTA Press.
- Smith, J. P., III, diSessa, A. A. & Roschelle, J. (1993–1994), 'Misconceptions reconceived: A constructivist analysis of knowledge in transition', *The Journal of the Learning Sciences* **3**(2), 115–163.
- Vegso, J. (2005), 'Interest in CS as a major drops among incoming freshmen', *Computing Research News* **17**(3).

# What's the Problem? Teachers' Experience of Student Learning Successes and Failures

**Arnold Pears**

Uppsala University, Sweden  
Arnold.Pears@it.uu.se

**Anders Berglund**

Uppsala University, Sweden  
Anders.Berglund@it.uu.se

**Anna Eckerdal**

Uppsala University, Sweden  
Anna.Eckerdal@it.uu.se

**Philip East**

University of Northern Iowa, USA  
east@cs.uni.edu

**Päivi Kinnunen**

Helsinki Uni of Tech, Finland  
pakinnun@hut.fi

**Lauri Malmi**

Helsinki Uni of Tech, Finland  
lma@cs.hut.fi

**Robert McCartney**

University of Connecticut, USA  
robert@engr.uconn.edu

**Jan-Erik Moström**

Umeå University, Sweden  
jem@cs.umu.se

**Laurie Murphy**

Pacific Lutheran University, USA  
lmurphy@plu.edu

**Mark Bartley Ratcliffe**

Valtech UK, London, UK  
mark.ratcliffe@valtech.co.uk

**Carsten Schulte**

Free University Berlin, Germany  
schulte@inf.fu-berlin.de

**Beth Simon**

UC San Diego, La Jolla, CA, USA  
bsimon@cs.ucsd.edu

**Ioanna Stamouli**

Trinity College Dublin, Ireland  
stamouli@cs.tcd.ie

**Lynda Thomas**

Uni of Wales, Aberystwyth, UK  
lth@aber.ac.uk

## Abstract

This paper opens the classroom door to provide insight into factors that shape tertiary computer science teachers' experience of (and engagement with) student learning success and failure. This topic is explored through phenomenographic analysis of teacher narratives dealing with frustration and success in facilitating learning for their students.

Three themes related to learning are explored which highlight different aspects of the learning situation, namely, *students*, *environment*, and *responsibility*.

Using these themes as a focus reveals great diversity in the manner in which teachers experience student learning difficulties and approaches to resolving them.

The results provide computer science academics with a framework within which to discuss and contrast their values and assumptions and understand their implications for teaching practice.

**Keywords:** computing education, student, learning, teacher experiences.

## 1 Introduction

Academic teaching life (in contrast to research life) is a largely private endeavour; what happens in the classroom is seldom discussed within the collegiate.

This paper is the outcome of a phenomenographic study (Marton 1976) of teacher experiences of student learning in computing. We also relate our findings to a wider discourse on how academics experience aspects of their teaching practice. The study outcomes describe teacher perceptions in relation to the success and failure of students in achieving learning goals. The nature of some of the categories of experience exposed in the study suggest that despite more than ten years of pedagogic development in universities, many academics experience helplessness in relation to assisting their students with learning difficulties.

Teaching and learning in higher education is increasingly important worldwide (Boyer 1997). General investigations that address student approaches to learning are well represented in the literature (Entwistle 1981, Biggs 1987, Booth 1992, Biggs 1999). Studies that focus on the role of the teacher are less common. Related results in higher education theory on teachers' approaches to teaching practice identify generic approaches applicable to a wide range of disciplines (Prosser & Trigwell 1999, Ramsden 1992, Biggs 1989, Martin et al. 2000, Trigwell & Prosser 2004, Trigwell et al. 1999). Earlier work on CS teachers' experience of their teaching demonstrates similar categories of approach (Lister et al. 2007).

This work differentiates itself from earlier studies in two ways. Firstly, in contrast to other more general results it focuses explicitly on *student learning difficulty/success*, describing how that phenomenon is experienced by CS academics. Secondly, it draws attention to a tension between the categories derived from the data and the claim that student-centred teaching and learning is the dominant approach in tertiary-level CS teaching.

	Category	Focus	Dominating aspect
		Student success is experienced as...	
1	The nature of the subject	being influenced by the inherent nature of the subject matter to be learned.	Subject
2	Intrinsic to the student	something neither the teacher nor the student can significantly influence.	Student
3	Prior experience	being due to education that prepares students for the current course.	Student
4	Attitude/behaviour	due to motivation, attitude and behaviour, e.g. study practices	Student
5	Developmental	a process of personal development with the help/guidance of the teacher.	Teacher and student

Figure 1: Experience of success

## 2 Data Collection and Analysis

Data for the study was collected as semi-structured interviews. Each author interviewed one to three colleagues (referred to as ‘teachers’ throughout the paper). A common interview script was used by all investigators. Sixteen interviews were collected for the study, describing teacher experiences of successful and unsuccessful attempts to facilitate learning of difficult CS concepts. The teachers interviewed were drawn from ten higher education institutions in seven countries (England, Wales, Ireland, USA, Sweden, Finland and Germany). Viewed as a collective these narratives provide a rich source of data for the study.

Initial analysis was performed in a two-day workshop attended by all the authors. The entire group worked intensively with the transcripts, trying to identify themes within the narratives. From this analysis three themes supported by rich data emerged. The subsequent analysis phases were performed in sub-groups associated with each of the three themes.

## 3 Results

In contrast to other studies we are concerned explicitly with CS teachers’ conceptions of success and failure and their experience of the nature of success or failure in CS studies. For each category of experience we also describe the focus (or nature) of the experience and its dominating aspect. To help the reader gain an impression of the type of data in the narratives that supports each category we provide illustrative citations from the full transcripts in the description of most categories.

### 3.1 Student

Categories that have a bearing on academics’ experience of what enables a student to succeed are shown in Figure 1.

#### 3.1.1 Category 1 – The nature of the subject matter

The most fundamental category excludes the student entirely. The focus is on the impact of the nature of the subject matter on students’ success. Narrative fragments relevant to this category focus on the inherent difficulty of specific topics such as algorithm analysis, pointers, concurrency and OOP. A quote from D1 serves to illustrate this:

D1: Because you know, pointers are less concrete than values. So they don’t always grasp that whole concept of the address. I just think they’re hard to visualise.

#### 3.1.2 Category 2 – Intrinsic to the student

In the second category success is experienced as an intrinsic feature of the student. Here success in learning seems to depend on a ‘magical’ feature (something that just is and for which the teacher has no concrete explanation). This can be a natural level of ability, innate way of thinking, aspect of gender, or a feature of culture and language. J1 expresses this type of understanding as follows:

J1: No, no, there are definitely some students that it just clicks [snaps fingers] with them. They have no trouble at all. They’ll be, and in fact, quite a few students, quite a few students just have a knack for that.

#### 3.1.3 Category 3 – Prior experience

In the third category, success is understood as something that is influenced by students’ background knowledge and prior experience. Difficulties in mastering computer science concepts are attributed to factors such as an inadequate mathematics background or lack of experience with fundamental skills such as logic or abstract reasoning. These types of deficiency appeared to be a source of frustration for teachers; as illustrated by the following excerpt:

D2: I think most haven’t thought too abstractly before they get into a CS program. Maybe if they took their math a long way as in high school, but even in high school, high school math is mostly, like, do the calc stuff and not logic and things like logic, and so I think most college students really haven’t had a lot of practice.

#### 3.1.4 Category 4 – Attitudes and behaviour

Another type of student-centred experience of success attributes the outcome to students’ own attitude or behaviour. In contrast to category 2, students’ attitude or behaviour is understood as mutable, something that a student can change. Several different attributes emerged from the data: effort and hard work, study habits, being proactive, presumptions, emotions and motivation.

The students’ own presumptions concerning CS, or their own inability to learn CS, are experienced as a

factor that hinders success. Students' motivation, both intrinsic and extrinsic, was seen as having an impact on success; for example, students' awareness of an upcoming assessment was recognised as being important. In the following quote O1 addresses the importance of hard work and persistence as fundamental precursors to success:

O1: Some students are very open to trying out that new thing. They know they don't get it yet, but they stick with whatever it is they're doing and they do get it. Whereas the person who tends to become overwhelmed either feels lost. Even if they want to stick with it they feel lost and they have a hard time. Or they turn away from it ...

### 3.1.5 Category 5 - Developmental

The focus in category 5 is on the strategies that a teacher uses to help students to succeed. Students' success is something that the teacher can influence: by presenting the material in multiple ways, using examples, practice exercises, pictures and manipulatives, assessing progress, and through fostering a positive learning context. Where differences in learning styles were experienced as being the source of difficulties, teachers attempted to help students by presenting the material in multiple ways:

A1: ... you have explained something a thousand times but then when you do it again then you realise that, Ah, okay! I have to come up with a new way of saying this because this student doesn't get my other ways.

## 3.2 Environment

### 3.2.1 Category 1 – Introspective/Self-centred

Category 1 describes the experience of a self-centric teacher. She or he does not engage with any external agent or resource. A variety of strategies are experienced by our interview cohort: passing the buck, ignoring student problems, reflecting on their own learning processes as a way to relate to student difficulties, changing the class requirements to remove difficulties, hoping that students will do better or complaining about student difficulties. S1 clearly experiences a need to displace responsibility:

S1: But I think the question is more what can be done with the given resources in time. Because that is the most limiting factor. Well, I'm quite sure that one can do, or: I could do a better teaching if I put more time in it. I mean I would have ideas how to do it. But that would just cost me a lot of time! However, I try to do the old 80-20 rule from managing: If you have to put 80% more effort to gain 20% more it's not worth doing it. So basically my approach for that is that I have a certain time-budget allocated for teaching and given that time I try to do the best [I can].

### 3.2.2 Category 2 – Environment

In this category, we meet experiences of ways to cope with students' difficulties by structuring the interaction environment in some way. Four variations of this category, differing in approach to handling the

environment, were identified. We present these distinct approaches as four sub-categories.

#### 2a) Preventing students from getting into difficulty.

One way to structure the environment is to prevent students from encountering a problem in the first place. This approach to structuring the learning situation differs from the others we observed, which explore teachers' coping with student difficulties after they are perceived. Here structuring is proactive and prophylactic, centred on setting up and maintaining student motivation, for example through connection to real-world applications. H1 expresses this in the following way:

Int: yeah, you mean that it's hard for them to see the connection to reality? [...]

H1: [...] they tend to, you know, shut their ears and they hope for some formula they can use and not really understand the theory behind it, eh, that is what I find hardest [...] To get them to, well, to, to, to find them interested in [...] But that is what I find hardest to, to really get them to find the motivation for it.

#### 2b) Expressing the material to the students.

Another way to structure the environment is by expressing things to the students, for example by doing examples, repeating material, telling students things, providing variation in material or explanations. Statements from I1 and D2 are illustrative here:

I1: And then I can provide some additional material, like lecture notes, I can provide lectures, I can provide the exercises, and [that] sort of stuff. So they can better cope with this material.

D2: Yeah. To try to present it in a bunch of different ways.

#### 2c) Directing the students to do something.

The environment can be structured by the teacher through directing the students to do something, with the goal of getting them to recognise or confront their difficulties. Variations include giving them assignments or activities to do (both in-class and outside of class), encouraging or even forcing them to do things. Let's listen to I1:

I1: Well, as I said before, I try to constantly improve my material, [...] I try to improve the exercises they have, [and] so on, so they actually face these difficulties or they face these concepts that they might have.

#### 2d) Engaging with the learner.

A fourth way of structuring the environment was to engage with learners in some way. Engaging with the learning differs from directing the learner in the sense that communication is bi-directional between teacher and student. But the goal of structuring the environment still remains focused on the students, interacting with them in the hope of getting them to recognise or experience a difficulty.

### 3.2.3 Category 3 – Extrovert

The previous two categories describe teacher-directed techniques to inform the student, through either the teacher's individual response to student difficulties or the

	Category	Focus	Dominating aspect
		When students encounter difficulties the teacher...	
1	Introspective	focuses inward, reflects on prior personal learning experiences and teaching behaviours.	Teacher
2	Environmental	changes the nature of the environment intending to prevent or remedy perceived difficulties.	Teacher
3	Extrovert	engages with students proposing activities to gain insight into students' experience.	Student and teacher

Figure 2: Experience of student difficulty

teacher's attempt to structure the environment. The goal was to guide students in order to prevent them experiencing or discovering difficulties. The third category differs in that it deals with situations where the teacher tries to understand student difficulties, with the implicit or explicit goal of using that information to assist the student. This can be achieved by monitoring, asking students to demonstrate code tracing and 'desk' execution, or seeking to understand how much the student knows. A1 here tells us about the latter experience as he prompts a student to trace execution in a code fragment:

A1: Let's say if this - if they are having problems with these details, for example, then I try to isolate where the particular detail is - yeah, what's wrong. And then I try to have them see it for themselves that this is where it is. And then, of course, if they still can't do it then I just say, "Well, okay. You have to have a semicolon after every line".

### 3.3 Responsibility

#### 3.3.1 Category 1 – Presenting the material

All, or nearly all, of the teachers state that they recognise and accept their responsibility of organising and presenting the subject matter well. This manifests itself in the transcripts as statements highlighting and presenting the important concepts, so that the students are aware of what they need to learn or where they need to put more effort. This is clearly voiced by J1:

J1: I think it's important, if there's a really important topic like that, I think it's important that I tell them this is a really important topic. And, you know, you're going to be tested on this and it's important that you understand this before you complete the course.

#### 3.3.2 Category 2 – Responding to students

The second category describes a perspective where teachers share responsibility for learning with the students. They expect students to identify the things they don't understand and act on those problems, for example, by asking questions, looking up other resources, etc. With this view, the responsibility for overcoming learning problems lies with both the student and the teacher, but the student is expected to take the initiative and voice his/her problems. Then the educator will further assist the student, as is here voiced by D2:

D2: Well, their job is to keep a positive attitude, I think, and keep trying, and not be resentful of not understanding it, and feeling like it's all my fault (the teacher's fault) if they're not getting it, and that ultimately, it's up to them to eventually get it,

and I'm there to help them, but they have to - you know, if they're not getting it, they have to try and express, rather than just saying, "I just don't get it." They need to think about [things] – formulate a question and then think what is it about it that I'm having trouble with, and so it helps me so that I can maybe better help them.

#### 3.3.3 Category 3 – Taking the student perspective

In the third category, the responsibility of the teachers is understood as viewing instruction from their students' perspective. A teacher can, for example, ask questions during the class to probe the students to assess their understanding. Let us listen to F1:

F1: Yes. Well the strategy is: if I have a formal topic and I see - so basically I ask - so after a couple of steps I ask questions that would make it clear whether the students have understood it or not. And I force somebody to answer. I just wait until you know until somebody answers - And I give them more and more hints which basically go more and more to the trivial and then I know they didn't get it and I repeat it and I phrase it differently or give a concrete example. And if I succeed you know if I have an answer and hopefully not only from the same student but from [a] different one.

This category clearly represents a more active approach to student learning problems and teaching in general. There is a sense of responsibility for identifying where students struggle, or at least for helping students identify what they do not understand, and for adapting teaching to better suit the students.

D2: I mean, my job is to not get frustrated with them, and to try and understand individually what it is, trying and see how they're thinking about it, and that can be hard. And try and understand how they're thinking about it so that maybe I can better get them to think about it differently.

## 4 Discussion

So what is successful student learning about?

We have identified five categories of experience related to students' ability to learn successfully. The active, student-centric, teacher ideal that much recent higher education development aims to produce is most strongly present in the fifth category.

Of greater interest in the light of staff development efforts are categories one and two in the current study. Clearly some teachers still experience student success or failure as not related to the student at all, or related to

	Category	Focus	Dominating aspect
		Teachers experience their responsibility as being to...	
1	Presenting the material	select, organise, and present material in a manner that facilitates student learning.	Content and teacher
2	Responding to students	in addition, encourage questions and respond to requests for assistance.	Student and teacher
3	Student perspective	respond as above, however, now teachers actively seek to understand what students have understood.	Student

Figure 3: Experience of responsibility

factors that are both inscrutable and impervious to change. For these individuals it is the subject matter, intrinsic properties of the student, or prior experiences that lead to successful learning outcomes. This view is ultimately disempowering for the teacher, since it implies that their efforts are ultimately futile. On the other hand, categories 4 and 5 present a less pessimistic view; here the teacher and the student can change things, affecting the learning process.

Teacher experiences related to successful learning outcomes clearly vary. A related picture can be obtained by examining teachers' experiences of the role of the environment in success or failure to facilitate student achievement. Figure 2 describes three distinct manners in which student learning behaviour is experienced. As in the previous discussion, the focus of the categories shifts from teacher to student as we move from category one to three.

When examining failure to learn, the second category is the most interesting as it encompasses four strategies a teacher might engage in while focusing on the learning environment. Two strategies are teacher-centric and aim to prevent failure and difficulty by restructuring the learning situation, a prophylactic approach to constructing learning situations. The other two strategies are centred on the students' learning challenging preconceptions about troublesome knowledge. One of these strategies is transmission-based while the other is bi-directional, aiming to engage students and help them to recognise and confront their difficulties.

Teachers' experiences of responsibility for learning are summarised in Figure 3. The focus on the content and organisation is perhaps not surprising given the import associated with course syllabi and other formal structural aspects of teaching. The experiences associated with interaction are also interesting. On one level they can also be interpreted in the light of the staff- or student-centric view of education. Only category 3 in figure 3 can be firmly associated with the student-centric model. There are also strong links to category three of Figure 2. It seems likely that teachers who have a more student-centric model of responsibility will also tend towards experiencing problems in the environment as extrinsic.

## 5 Conclusion

While much effort has been spent on raising academics' awareness of student-centric approaches to teaching and learning in computer science over the last ten years, it appears that teacher attitudes lag behind. Teachers experience disempowerment in the conduct of their day-to-day teaching practice and there is a risk that this can lead to passivity and disengagement, to the detriment of learning.

## References

- Biggs, J. (1989). Approaches to the enhancement of tertiary teaching. *Higher Education Research and Development* 8(1):7-25.
- Biggs, J. (1987). *Student Approaches to Learning and Studying*. Australian Council for Educational Research, Melbourne, Australia.
- Biggs, J. (1999). What the student does: teaching for enhanced learning. *Higher Education Research and Development* 18(1):57-75.
- Booth, S. (1992). *Learning to program. A phenomenographic perspective*. No. 89 in 'Göteborg Studies in Educational Science', Actae Universitatis Gothoburgensis.
- Boyer, E. (1997). *Scholarship Reconsidered: Priorities of the Professoriate*, Jossey-Bass, Hillsdale, NJ.
- Entwistle, N. (1981). *Styles of teaching and learning: an integrated outline of educational psychology*. John Wiley, Chichester.
- Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., Bower, M., Carbone, A., Davey, B., de Raadt, M., Doyle, B., Fitzgerald, S., Manilla, L., Kutay, C., Peltomäki, M., Sheard, J., Simon, Sutton, K., Traynor, D., Tutty, J. & Venables, A. (2007). Differing Ways that Computing Academics Understand Teaching. *Australian Computer Science Communications*, 29(5):97-106.
- Martin, E., Prosser, M., Trigwell, K., Ramsden, P., & Benjamin, J. (2000). What university teachers teach and how they teach it. *Instructional Science* 28(5):387-412.
- Marton, F. & Säljö, R. (1976). On qualitative differences in learning. *British Journal of Educational Psychology* 46:4-11 & 115-127.
- Prosser, M., & Trigwell, K. (1999). *Understanding Learning and Teaching: The experience in higher education*. Open University Press, Buckingham.
- Ramsden, P. (1992). *Learning to teach in higher education*. Routledge, London.
- Trigwell, K. & Prosser, M. (2004). Development and use of the approaches to teaching inventory. *Educational Psychology Review* 16(4).
- Trigwell, K., Prosser, M. & Waterhouse, F. (1999). Relations between teachers' approaches to teaching and students' approaches to learning. *Higher Education* 37:57-70.



# Computer Science students can help to solve problems of multiplayer mobile games

Carolina Islas Sedano

Ekaterina Kuts

Erkki Sutinen

Department of Computer Science and Statistics,

University of Joensuu

PO. Box 111. FIN-80101, Joensuu

{cisl, ekuts, sutinen}@cs.joensuu.fi

## Abstract

This study analyzes the possibility of involving Computer Science students in multiplayer mobile game development processes and using their knowledge to solve collaboration problems. We describe the importance of Computer Science as one of the Computing disciplines, and provide a summary of its knowledge areas according to their ability to help solve collaboration problems. This analysis is based on the Bachelor's degree study program at the University of Joensuu, Finland. Research shows that it is highly possible to involve students from their first year of study in solving collaboration problems of multiplayer mobile games.

**Keywords:** education mobile games, collaboration, communication

## 1 Introduction

Some researchers assert the need to improve the quality and impact of studies in the fields of education and technology (Reeves 2006). One of the possible solutions is to improve the research design based on development goals. Thus, the activity should be focused on creative approaches to solving problems while at the same time constructing reusable design principles. The development of games with educational purposes clearly falls within this categorization. As a consequence both objectives are constantly present in games.

On the other hand, mobile technology is a new field of research. It offers the opportunity to embed learning within the environment (Schwabe & Göth 2005). Mobile devices have a number of unique advantages, such as accessibility for general use, a broad reach, a wide spectrum for executing tasks and constantly growing capabilities. In particular these functions enable us to use mobile devices in education (Mitchell et al. 2007).

Combining game development with the use of mobile technologies offers us a wide set of possibilities. In addition, the learning experience can be enhanced due to the mobility and agility of participants in mobile games.

Computer Science has an important role to play in the growth of mobile games in general, regardless of educational aspects. Computer Science will not only help with the understanding of the technology but will also foster the formation and creation of interdisciplinary knowledge. Nevertheless, it is important to remember that any attempt to produce something by interdisciplinary means implies that as the complexities increase in the environment, so they do in the outcome. In other words, if game design and educational technologies already present challenges, once we combine them the complexities multiply. However, dealing with new approaches and technologies allows one to open a way for collaboration between disciplines. While this is a new phenomenon for the fields involved, it improves the possibilities of success. Mobile games can be one potential step towards this success.

Mobile games are the most popular applications for mobile devices (Wagner 2005). Notwithstanding their advantages, educational multiplayer mobile games have a set of problems. Most of them have shortcomings in technological areas. We cannot avoid the fact that while the development of mobile devices is improving rapidly, devices with strong limitations are still in use. Hardware problems include slow CPU speed, limited storage space, low precision on screen and the requirement of a large battery capacity for gaming purposes. Besides hardware challenges, mobile games also reveal problems in game play, personal understanding and software implementation.

Focusing on educational multiplayer mobile gaming, it is essential to understand the communication and collaboration between learners or players (Antonellis et al. 2005). This knowledge is fundamental for further mobile game development not only with an educational purpose but in any type of mobile game.

In this paper, after reviewing the literature, we observe the technology and communication trends that are presented in educational mobile multiplayer games, and their attendant challenges. In response to these challenges, we perform an analysis within the Computer Science discipline, searching to understand how this field can improve or solve communication problems. This is important for proper collaboration, and it also, in some cases, offers possible techniques for solving problems. The main criterion in the analysis of those techniques is an experience of solving similar problems in other areas. We are interested in ascertaining **how students of computer science can help to resolve some problems**

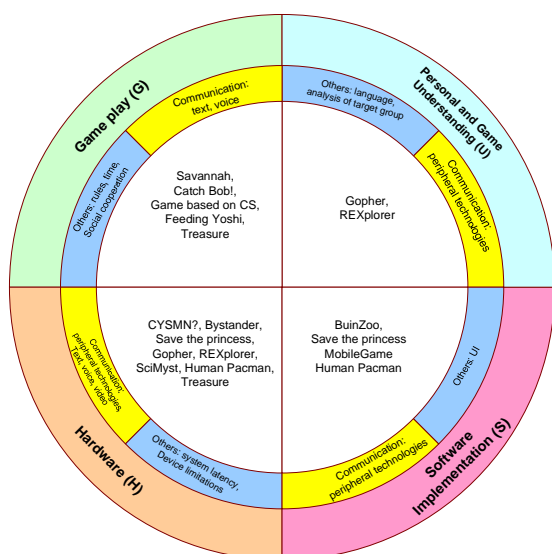
**presented in educational mobile games and at the same time gain knowledge in related subjects of the discipline.** Furthermore, we investigate how this can be integrated into the existing curriculum.

Consequently, the scope of this paper includes the overview of existing educational multiplayer mobile games and their challenges. The next section contains an analysis of the potential of Computer Science to solve existing problems at the University of Joensuu, Finland. We continue with a section in which we discuss possible ways to engage students in this area.

## 2 Mobile game collaboration problems

In our search of the common challenges in educational multiplayer mobile games we first reviewed published papers about existing projects or their prototypes. We identified 16 games that displayed both an educational and a multiplayer component (Kuts et al. 2007).

Further analysis allowed us to find the communication types used today, among them private and public text, photo and video message exchange, calls, and game peripheral technologies. We define peripheral technologies as technologies that provide technical information from a game server about other players' positions or actions to use the features of the modern mobile device. Considering collaboration as cooperation and integration of participants' intellectual facilities, it is possible to analyse the main challenges through the design of collaboration actions in the reviewed material. We observed problems with game play (G) (e.g. coordination problems), personal and game understanding (U), hardware problems (H), and software implementation problems (S). Each group of challenges



**Figure 1: Collaboration problems in educational multiplayer mobile games**

was divided into two parts according to its type of communication. We visualize the data in Figure 1.

## 3 Computing science potential helper

Eric Roberts, co-chair of ACM's Education Board and professor of Computer Science at Stanford University, mentioned that "*almost every major challenge facing our world is turning to computing for a solution, from conquering disease to eliminating hunger, from improving education to protecting the environment.*" (Gold et al. 2007). It cannot be done just by computers per se. All those operations need human knowledge to use computers properly for specific purposes. However, from a Computer Science perspective one needs to have solid knowledge and skills that will allow one to implement technologies to resolve specific problems. Some of the basic knowledge is suggested by the major computing disciplines: Computing Engineering, Computer Science, Information Systems, Information Technology and Software Engineering (Computing Curricula 2005). In this classification Computer Science covers a wide range of areas, from theoretical research to different development tasks. In this case, gaming is an even more complex multidisciplinary field, which includes computer science, information technologies, art and media.

Year 1			
term 1	Introduction to Computer Science (175111)	Programming (175112)	IT passport
term 2	Programming (175112)	Discrete Structures (175114)	
term 3	Programming (175112) Laboratory Project on Programming (175113)	Computer Systems (175115)	
term 4	Data Structures and Algorithms 1 (175211)	Human Factors of Interactive Technology (175214)	
Year 2			
term 1	Data Management (175212)	Procedural Programming (175213)	
term 2	Laboratory Project in Computer Science (175215)	Freely elective courses	
term 3	Data Structures and Algorithms 2 (175217)	Scientific Writing in Computer Science (175131)	
term 4	Distributed and Concurrent Systems (175219)	Scientific Writing in Computer Science (175131)  Freely elective courses	
Year 3			
term 1	Introduction to Software Development (175220)	Software Project (175226)	
term 2	Freely elective courses	Software Project (175226)	
term 3	Theory of Computing (175221)	Bachelor's degree thesis (175291)	
term 4	Freely elective courses	Bachelor's degree thesis (175291)	

**Table 1: Discipline-based model**

Multiplayer mobile games collaboration problems			CS knowledge areas
Game Play (G)	Direct	<b>Voice and text.</b> For games based on voice communication the frequent problem is a lack of understanding. This later leads to problems in strategy formulation and players' confidence.	Computer System Engineering, System Integration and HCI
	Indirect	<b>Synchronization.</b> The synchronization that should take place is between partners or a team, but also outside them. <b>Time.</b> To achieve stable player cooperation and collaboration, the players have to spend some time playing together, but normally mobile games do not offer enough time for it. <b>Rules.</b> For some games cooperation is not required, and you can reach the game goal by playing alone. In other situations game rules can be unclear or too difficult.	Digital Logic and HCI
Personal and Game Understanding (U)	Direct	<b>Peripheral technologies.</b> Technological misunderstanding for the players. Some games require special equipment and for some participants it can be problematic to use new devices.	HCI
	Indirect	<b>Social characteristics.</b> The difference in education level, age, social background, or incorrect analysis of target groups bring challenges for the understanding of the game or devices. It can also present ethical problems.	Legal / Professional / Ethics / Society
Hardware related Problems (H)	Direct	<b>Text, voice, video, peripheral technologies.</b> System latency is one of the most limiting factors for communication problems. E.g. mobile providers do not support mostly video streams, or the implementation of voice conferences on mobile devices can be also complex. These can affect the players' communication and learning level.	Programming Fundamentals, and Computer Architecture
	Indirect	<b>Device limitations.</b> Slow CPU speed, limited storage space, low precision on screen and requirement of large battery capacity.	
Software Implementation (S)	Direct	<b>Peripheral technologies.</b> Some games have implementation problems as they do not sufficiently enable hardware features within software, or tools for collaboration support.	Software Modeling and Analysis, Software Design, Software Verification and Validation, Software Evolution (maintenance), Software Process and others
	Indirect	<b>User interface.</b> Some games present insufficient or incorrect icons, color, characters between other interface features. In some cases it is not clear how to use game functions or navigate through the phones. <b>Visualization.</b> In location-based games, player does not always see new position on a screen. Other type of games do not visualize what the player is typing. For some players, written communication is difficult even in PDA.	Graphics and Visualization

Table 2: Collaboration challenges and knowledge areas

To find disciplines that can help to solve the collaboration challenges of multiplayer mobile games we analysed the study curricula at the University of Joensuu. Table 1 shows disciplines according to the year of study. We observed obligatory disciplines for all students in the department of Computer Science and Statistics for the Bachelor's degree.

Turning next to the ACM Computing Curricula, we emphasize knowledge areas that can be useful for multiplayer mobile games, as shown in Table 2.

Now it is possible to analyse disciplines at the University of Joensuu according to their challenges and knowledge area. In this way we formulated Figure 2, which allows us to understand the study program, and find combinations of knowledge and experience necessary for the exact game project. Freely elective courses might provide knowledge to solve some of the problems, but they are beyond the scope of this paper. We find that we can involve different students in multiplayer mobile game development processes from their first year of study.

#### 4 Discussion

The idea of involving students in multiplayer mobile game development is relevant today. It can help students to improve their knowledge not only in Computer Science, but in related fields. We want to emphasize that educational mobile games can offer learning experiences from an early developmental stage. For example, first year students can participate in group work even without deep knowledge in the subject, but offering them a foretaste of where they can go with their education. In this case, they have to solve small tasks or develop some features that allow them to work and study together with game developers and computer science experts. Additionally, with the works on offer and the ongoing university game projects, at the end of the development students can play with their product, and this might increase the likelihood of engaging people in the study.

Moreover, this developmental route aggregates learning, including experience in related subjects, and also offers enjoyment while conquering challenges to reach goals; which is clearly desirable for students.

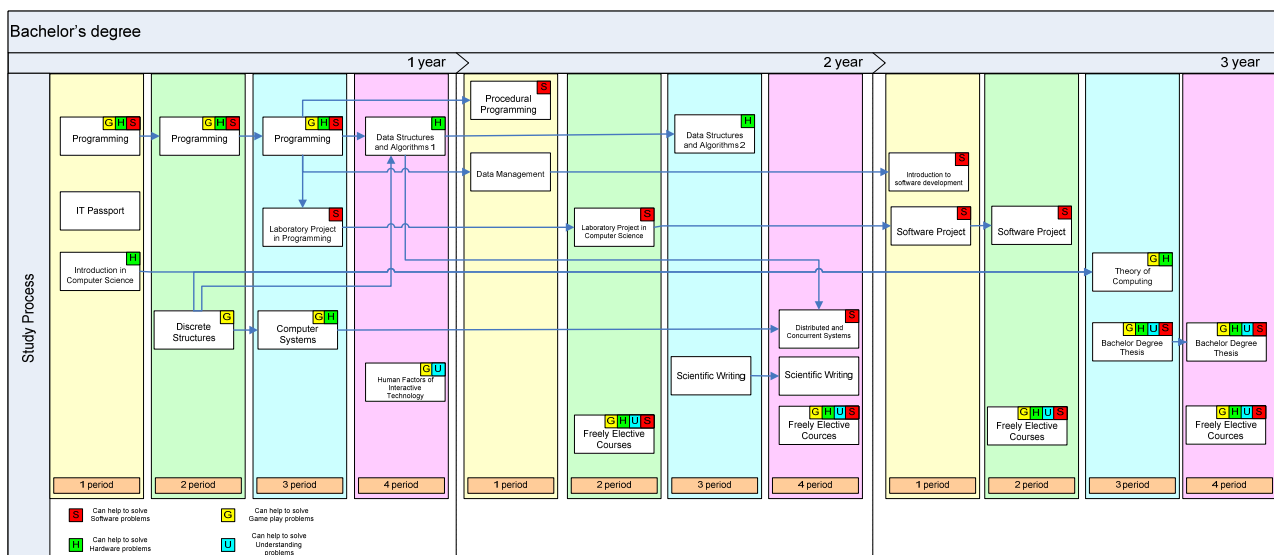


Figure 2: Bachelor's degree study plan and opportunity to use students' knowledge

According to previous findings (Table 2 and Figure 2) we can compose Figure 3, which shows by year of study the number of disciplines that can help to solve challenges in multiplayer mobile games.

We see that it is in the first year of study that students get most of the knowledge needed to help solve collaboration problems in educational mobile games. Furthermore, students' attraction to the game development process from the beginning of the project and of their university study can help not only to improve knowledge but to understand further career preferences and to work together with competent people.

## 5 References

- Antonellis I., Bouras C., Pouloupoulos V. (2005): Game Based Learning for Mobile Users. 6th International Conference on Computer Games: *AI and Mobile Systems* (CGAIMS 2005), Louisville, Kentucky, USA.
- Computing Curricula 2005: The Overview Report, 30/09/2005. ACM Press. URL: [http://www.acm.org/education/curric\\_vols/CC2005-March06Final.pdf](http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf). Accessed July 28, 2007
- Gold, V., Wilson, C. (2007): Guide to Computing Careers Helps Students Develop Right Skills. ACM Press Room. URL: <http://www.acm.org/press-room/news-releases/computingcareers/>. Accessed July 28, 2007.

- Kuts, E., Islas Sedano, C., Sutinen, E. (2007): Communication and collaboration in educational multiplayer mobile games. *Proceedings of Cognition and Exploratory Learning in Digital Age*. December 7-9, 2007. Algarve, Portugal. (forthcoming)
- Mitchell, A., Csic, D., Maxl, E. (2007): Mobile Game-Based Learning – issues emerging from preliminary research and implications for game design. URL: [http://domino.fov.uni-mb.si/proceedings.nsf/Proceedings/D0C83B3C486A940AC12572EE007AD D88/\\$File/Paper105.pdf](http://domino.fov.uni-mb.si/proceedings.nsf/Proceedings/D0C83B3C486A940AC12572EE007AD D88/$File/Paper105.pdf). Accessed July 13, 2007.
- Reeves, T. (2006): IT Design Based Research. Saving Instructional Technology from Irrelevance: The Promise of Design Research. URL: <http://www.uga.edu/greip/events-it-design-based-research.html>. Accessed August 10, 2007.
- Schwabe, G. and Göth C. (2005): Mobile Learning with a Mobile Game: Design and Motivational Effects. *Journal of Computer Assisted Learning*, vol. 21, no. 3, pp. 204.
- Wagner, E.D. (2005): Enabling Mobile Learning. *EDUCAUSE Review*, vol. 40, no. 3 (May/June 2005): 40–53. URL: <http://www.educause.edu/apps/er/erm05/erm0532.asp>. Accessed July 21, 2007

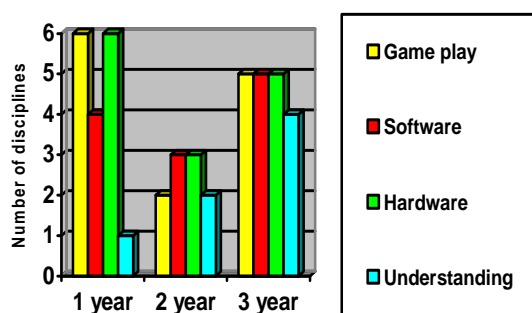


Figure 3: Approximate number of disciplines by year of study that can help to solve problems

# Applications of variation theory in computing education

**Jarkko Suhonen**

Department of Computer Science and Statistics  
University of Joensuu  
PO. Box 111  
FI-80101 Joensuu  
Finland

[jsuhon@cs.joensuu.fi](mailto:jsuhon@cs.joensuu.fi)

**Errol Thompson**

2 Haven Grove,  
Lower Hutt,  
New Zealand

[kiwiet@computer.org](mailto:kiwiet@computer.org)

**Janet Davies**

School of Curriculum and Pedagogy  
Massey University  
Private Bag 11 222,  
Palmerston North  
New Zealand

[j.r.davies@massey.ac.nz](mailto:j.r.davies@massey.ac.nz)

**Kinshuk**

Professor and Director  
School of Computing and Information Systems  
Athabasca University,  
1 University Drive  
Athabasca, Alberta T9S 3A3  
Canada

[Kinshuk@ieee.org](mailto:Kinshuk@ieee.org)

## Abstract

There are various challenges in IT education. Students have difficulties in applying the key concepts, theories and techniques taught in introductory courses. At the same time, the backgrounds of students are becoming more and more diverse. Teachers work with heterogeneous student cohorts. Phenomenographic studies can be used to understand different perspectives of learners' understanding. Variation theory is a promising approach to improving the teaching of computing subjects. The theory can be applied to design variations in teaching that make explicit the different aspects of computing concepts. We discuss the role of variation theory in creating diversity in teaching practices to reach students with diverse backgrounds and expectations.

*Keywords:* variation theory, programming.

## 1 Introduction

Research has indicated that there are universal challenges in the teaching and learning of computing subjects. Learners are not reaching the desired performance levels and they have problems even with the basic concepts, especially after the first programming courses (McCracken et al., 2001, Lister et al., 2004a). Novice programmers, in particular, seem to have a wide range of

difficulties and deficits (Robins et al., 2003). Students in computing subjects now have more varying backgrounds, skills, and motivations. Students might have been used to a range of teaching and learning approaches. Furthermore, students' behaviour patterns can also be different, which causes problems with communication, collaboration and interaction. Taking these factors into consideration, there is a need for novel pedagogical approaches in the CSE community to fulfil the needs of heterogeneous students. In this paper we discuss the applicability of variation theory as a course design tool to enrich teaching practices in computing subjects. We provide two examples of the use of variation theory and discuss the role of variation theory in assisting teachers to reach students with diverse backgrounds and expectations.

## 2 Application areas of variation theory in teaching and learning

### 2.1 What is variation theory?

The foundation of variation theory is based on the concept that people become aware of a phenomenon through the way that it varies from its environment (external horizon) or the way in which its internal parts vary in relation to one another (internal horizon) (Booth, 1992, Marton et al., 1993, Marton and Booth, 1997). When considering the nature of variations, there are those that on the surface seem obvious and those that are highlighting significant details. Variations help in identifying a phenomenon but there are also variations in the way that individuals recognise or are aware of a phenomenon. These variations in the ways in which people are aware of a phenomenon are used in a phenomenographic study to develop *categories of description*. It has been shown that there are a limited number of categories of description or ways in which people are aware of a phenomenon (Marton and Booth,

1997, Marton, 2000). These categories of description are usually placed in a hierarchy, where each higher layer incorporates or expands on the previous description. It is argued that learning occurs when a person becomes aware of a phenomenon in a different way.

*Variation theory* focuses on the way that a phenomenon is made visible in a teaching context (Marton and Tsui, 2003). Utilising knowledge of the variations with respect to how the phenomenon stands out from other things in the environment and with respect to the phenomenon's internal structure, it is possible to focus on the aspects that will help build the desired level of understanding. Marton et al. (2003) have defined the patterns of variations that are considered to be significant:

**Contrast:** “in order to experience something, a person must experience something else to compare it with” (p16). This may be a way of identifying critical aspects of the phenomenon with respect to other phenomena.

**Generalisation** is required with respect to the object of learning (p16). It isn't enough simply to see the object; we need to see variations in the use of the object to fully comprehend it. This involves recognising that some features are not critical to the identification of that phenomenon.

**Separation** of an aspect from other aspects is required. The object needs to be looked at from different angles. The aspect being examined “must vary while other aspects remain invariant” (p16).

**Fusion** is where “several critical aspects” need to be considered together. Those aspects must be experienced simultaneously (p16).

## 2.2 Variation theory in planning teaching

A core idea behind variation theory in the teaching context is that the teacher develops the teaching material with a perception of the content, that is, an “*intended object of learning*”. Marton et al. (2003) argue that the object of learning is defined by “its critical features, that is, the features that must be discerned in order to constitute the meaning aimed for” (p22). A *critical feature* is a way of “distinguishing one way of thinking from another” (p24). In terms of an object of learning, variation theory differentiates effective and ineffective ways of experiencing that object. From these, it is then possible to look at the variations that can be used to ‘enact’ the phenomenon by using the patterns of variation or categories of description (Cope and Prosser, 2005).

The teacher can use appropriate variations within the identified space of learning to enact the object of learning (Marton et al., 2003). To create a space of learning there is a need to open up a dimension of variation. This variation may be through contrast with other objects in the learning environment or through variations in the frame of reference or ways of looking at the object of learning. The range of what can be learnt is defined by the space of learning and the range of variations presented in the learning space. The space of learning equates to an experience space in that it opens up ways of experiencing the object of learning (p25). One

perspective of creating appropriate spaces of learning is to recognise that learners' previous experiences and knowledge influence their learning approaches and outcomes (Ramsden, 2003). If a teacher is aware of these variations, she or he can use examples, exercises or teaching approaches that are culturally relevant and familiar to the students. This increases the opportunity for the students to become aware of alternative ways of seeing the phenomenon. The linking with the learner's background is especially important with multi-cultural learner groups. Bates (1999) reports, for instance, that examples, idioms and writing styles may not be easily transferred between cultures. Learners can also have very different expectations of interaction patterns between instructors and students. For instance, in a western educational culture teachers often emphasise critical thinking skills, debate and discussion, while students from other cultures might have greater respect for the teacher and written text. In multi-cultural student cohorts it could be useful to raise the awareness of what learning is about and have some discussion about various ways of understanding the object of the study from different perspectives. For instance, the students' current conceptions and thinking processes can be uncovered through counter-examples or variations. As a result, learners' prior knowledge structures may change radically, when possibly existing naïve ideas and concepts have to be exchanged for viable knowledge (Bolhuis, 2003).

Important aspects of a learning domain are ‘critical aspects’. When designing a course, it is the variations with respect to the critical aspects that help the learner to become aware of the phenomenon in the desired way (Cope, 2000). Critical aspects are identified as topics that are conceptually difficult, alien and/or counter-intuitive for students. Recent research results indicate that the critical concepts in the programming domain are object-oriented structures, pointers, recursion and procedural abstraction (Jenkins, 2001). Failing in the critical aspects will likely block learners' future progress and eventually lead to frustration and even to abandonment of studies.

What the student takes away from a learning situation as an understanding is the “lived object of learning” (Marton et al., 2003). Studies among computing themes have explored student conceptions of the phenomenon being studied (Booth, 1992, Berglund, 2002, Lister et al., 2004b, Eckerdal & Thuné, 2005). These studies help highlight the variations in perception that the students have as a result of the learning or the lived object of learning. In analysing the effectiveness of teaching, we can endeavour to assess how the intended, enacted, and lived objects of learning compare (Boyer, 1990, Glassick, 2000). The intended object of learning can be compared to the categories of description relevant to the phenomenon in order to determine both the level of awareness being focused on and the appropriateness of the intended object of learning. This could be considered an initial assessment of whether the teaching is intended to target the appropriate level of learning. The intended and enacted objects of learning can be compared to determine whether what is being taught matches what

was intended to be taught. The student's lived object of learning can be compared against categories of description as a means of assessing the level of learning achieved or against the enacted level of learning to determine whether the enacted object of learning is being transferred to the lived object of learning as expected.

### 3 Examples of variation theory in computing education

#### 3.1 Programming concepts

In teaching programming, there are at least two dimensions. There are the constructs or concepts that are to be introduced and the problems that those constructs or concepts are used to solve. By using variations in problems for a given construct or concept, we can emphasise the characteristics of that construct or concept within the context of the program. The construct or concept remains unchanging but the context varies. An alternative approach is to focus on a consistent problem but apply variations in the use of a construct or concept, or apply different constructs or concepts to the solving of that problem.

A fundamental concept in learning programming is the concept of a program. To start exploring the question 'What is a program(me)?', a teacher might present several different examples of programmes that are familiar to the student. This provides a link to prior knowledge. These might include TV programmes, computer programs, recipes, building instructions, and instructions for roles in a restaurant (Thompson, 2006). The student would then discuss what the characteristics are that make these examples of programmes. The initial conception of a programme envisaged by the teacher is that of algorithm plus data (Wirth, 1976). Later examples, such as the restaurant example, challenge this thinking to focus on the idea of interacting entities or entities that respond to specific requests (Stein, 2003)<sup>2</sup>.

The conceptual understanding of students can be further strengthened by comparing programme examples with non-programme examples and/or by letting students come up with their own examples of programmes and non-programmes. Further discussion can be held to investigate the characteristics of programmes. A goal of this activity is to discuss the key aspects and to construct a conceptual understanding of a programme and how it might be created. The examples are chosen to highlight the characteristics of a computer programme represented by the teacher's 'intended object of learning'. They are also chosen because they represent variations of the concept

within the context familiar to the students (background knowledge).

#### 3.2 Software development process

An approach to address the issues related to teaching of software development is to model the software development strategy. For example a strategy such as 'divide and conquer' or 'programming by intent' could be applied to a range of problems and applied consistently through the teaching of concepts and ideas. In this case, the process remains static but the problems vary, thus revealing that the process is not problem-dependent. If the opportunity were available, it might also be desirable to vary the programming language or development environment so that the learner saw that these strategies for programming are not dependent on the programming language or programming paradigm. They may be adapted but still utilised.

An alternative variation strategy is to use different software development strategies to solve the same problem. For example, the strategy of 'functional decomposition' might be contrasted with 'programming by intent', thus allowing the learner to see that alternative strategies can be used to address the same problem and possibly come to similar solutions.

With respect to the learning processes of software development, instead of modelling a software development strategy it would be necessary to model a process of discovery and exploration of domain concepts and of programming language constructs. This might include strategies for exploring framework features; for example, developing a series of automated tests to build an understanding of a framework object or feature. Again the modelled process could be applied to a range of learning issues within software development, including explorations for understanding the domain and for understanding the tools being used in software development.

### 4 Discussion and Conclusion

In this paper we have discussed the applicability of variation theory to the design of teaching with specific emphasis on computing subjects. We have endeavoured to emphasise how variation theory can be used to strengthen the understanding of a phenomenon by using variations relating to its critical aspects to form the space of learning. It has been suggested that knowledge of these critical aspects can assist in the planning process and that knowledge of possible variations can assist the teacher in helping students with their learning.

It is easy to argue that variations are always used in teaching. The question is whether those variations are effective. From a variation theory perspective, the variations that are effective are those that help present the critical aspects of a phenomenon and not simply those that show how the phenomenon varies from some other phenomenon. If we revisit the 'What is a program(me)?' example, what worked were the variations that helped students to think about algorithm and data. They were presented with a number of familiar and maybe not so

---

2 The use of the British spelling of programme in this context is deliberate and is intended to provide a link with the programme concept outside of the computing world. This is to enhance the linkage with the student's background knowledge.

3 Note research is currently under way to gain a better understanding of the conceptions of an object-oriented program.

familiar program(me)s that focused on these two aspects. They were also given non-examples that focused either on the data or on the algorithm but did not include both. The examples also avoided including other complexities of programming such as syntax and semantics.

If variations are to be used in teaching then it isn't adequate simply to know the definitions and to be able to present these to the students. The teacher needs to understand the critical aspects that are encapsulated in those definitions and the variations of examples and questions that will help create an enacted object of learning that makes those critical aspects visible.

When a course is evaluated using variation theory, one looks for those variations that highlight the critical aspects. It is these that are going to define the enacted object of learning and create the space of learning. It therefore seems logical, if we are to use variation theory in planning teaching, that the course designer and teacher understand what these critical aspects are and the variations that will make them visible to the learner. Future research could be focused, for instance, on investigating what kinds of variation would be especially useful in computing subjects.

## 5 References

- Bates, T. (1999) Cultural and ethical issues in international distance education. *Proceedings of the Engaging Partnerships Collaboration and Partnership in Distance Education UBC/CREAD conference*. Vancouver, Canada.
- Berglund, A. (2002) On the understanding of computer network protocols. *Department of Information Technology*. Uppsala, Sweden, Uppsala University.
- Bolhuis, S. (2003) Towards process-oriented teaching for self-directed lifelong learning: a multidimensional perspective. *Learning and Instruction*, 13, 327-347.
- Booth, S. A. (1992) *Learning to program: A phenomenographic perspective*, Göteborg, Acta Universitatis Gothoburgensis.
- Boyer, E. L. (1990) Scholarship reconsidered: priorities of the professoriate, San Francisco, Jossey-Bass Inc.
- Cope, C. (2000) Educationally critical aspects of the experience of learning about the concept of an information system. *School of Arts and Education Faculty at Bendigo*. Bundoora, Victoria, La Trobe University. From Education/Phenomenography/Cope2000Thesis.pdf.
- Cope, C. & Prosser, M. (2005) Identifying didactic knowledge: An empirical study of the educationally critical aspects of learning about information systems. *Higher Education*, 49, 345-372.
- Glassick, C. E. (2000) Reconsidering scholarship. *Journal of Public Health Management and Practice*, 6, 4-9.
- Eckerdal, A. & Thuné, M. (2005) Novice Java programmers' conceptions of "object" and "class", and variation theory. *ITiCSE'05*. Monte de Caparica, Portugal, ACM. From LearningToProgram/VariationTheory/EckerdalThune 2005ITiCSE.pdf.
- Jenkins, T. (2001) The motivation of students of programming. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*. Canterbury, UK, ACM Press.
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004a) A multi-national study of reading and tracing skills in novice programmers. *Inroads - The SIGCSE Bulletin*, 36, 119-150.
- Lister, R., Box, I., Morrison, B., Teneberg, J. & Westbrook, S. (2004b) The dimensions of variation in the teaching of data structures. *Inroads - The SIGCSE Bulletin*, 36, 92-96.
- Marton, F. (2000) The structure of awareness. In Bowden, J. A. & Walsh, E. (Eds.) *Phenomenography*. Melbourne, Australia, RMIT University Press.
- Marton, F. & Booth, S. A. (1997) *Learning and awareness*, Mahwah, NJ, Lawrence Erlbaum Associates.
- Marton, F., Runesson, U. & Tsui, A. B. M. (2003) The space of learning. In Marton, F. & Tsui, A. B. M. (Eds.) *Classroom discourse and the space of learning*. Mahwah, NJ; London, Lawrence Erlbaum Associates, Publishers.
- Marton, F. & Tsui, A. B. M. (Eds.) (2003) *Classroom discourse and the space of learning*, Mahwah, NJ; London, Lawrence Erlbaum Associates, Publishers.
- McCracken, M., Kolikant, Y. B.-D., Almstrum, V., Laxer, C., Diaz, D., Thomas, L., Guzdial, M., Utting, I., Hagan, D. & Wilusz, T. (2001) A multi-national, multi-institutional study of assessment of programming skills for first-year CS students. *Inroads - The SIGCSE Bulletin*, 33, 125-140.
- Ramsden, P. (2003) *Learning to teach in higher education*, London, Routledge Falmer.
- Robins, A., Rountree, J. & Rountree, N. (2003) Learning and teaching programming: A review and discussion. *Computer Science Education*, 13, 137-172.
- Stein, L. A. (2003) Interactive programming in Java. From [http://www.cs101.org/ipij/ObjectOriented/Interactive ProgramminginJava](http://www.cs101.org/ipij/ObjectOriented/InteractiveProgramminginJava).
- Thompson, E. (2006) What is a program(me)? From [http://www.massey.ac.nz/~elthomps/What\\_is\\_a\\_program/programmes.html](http://www.massey.ac.nz/~elthomps/What_is_a_program/programmes.html).
- Wirth, N. (1976) *Algorithms + data structures = programs*, Englewood Cliffs NJ, Prentice Hall.

# How Does Internationalisation Affect Learning and Teaching of Computer Science: A Study at Tongji University in China

**Doris Dongsheng Yang**

School of Software Engineering  
Tongji University  
Shanghai  
P. R. China

dorisyang@mail.tongji.edu.cn

Also affiliated to:

Faculty of Education  
The University of Hong Kong  
Hong Kong

**Anders Berglund**

Department of Information Technology  
Uppsala Computing Education Research Group,  
UpCERG  
Uppsala University  
Uppsala, Sweden

Anders.Berglund@it.uu.se

Currently also affiliated to:

Helsinki University of Technology  
COMPSER laboratory  
Department of Computer Science and Engineering  
Espoo, Finland

## Abstract

In the era of globalisation, higher education institutions cannot be excluded from internationalization. It has become a main theme of universities all over the world, including China. This discussion paper describes a proposed research project exploring how internationalisation in a Chinese university affects students' learning of computer science and the quality of teaching. It opens for discussion the question of what a Chinese university can learn about computer science education from international collaborations. Three questions are proposed for discussion on the focus of the research project and its methodology.

**Keywords:** internationalisation, computer science education, computing education, teaching quality, learning environment.

## 1 Introduction

With the current development of ICT and globalisation, every part of the world is closely linked to every other. No country can remain isolated from the rest of the world. In order to prepare our university graduates to perform and work effectively and efficiently in this globalised arena, it is important to help them to learn to interact and work collaboratively with counterparts who have different cultures and backgrounds. In this way, we hope that their horizons will be broadened and their capabilities for dealing with issues involving inter- or multi-cultural perspectives will be enhanced.

Two questions concerning internationalisation of the Chinese academic environment have recently inspired the first author of this paper, Doris Yang, to submit a research proposal to University of Hong Kong, Hong Kong. The questions are:

- How does internationalisation in a Chinese university affect how the students learn computer science?
- What can a Chinese university learn about computer science education from international collaborations?

In this paper we will elaborate on some aspects of this forthcoming project, with the aim of discussing the values of internationalisation in computer science education, as well as possible ways to gain insights about these values. The paper starts with a discussion of the significance of the upcoming project (section 2), continues with a literature review (section 3), then presents the project with its theoretical underpinning (section 4), and finally proposes questions for discussion (section 5).

## 2 Significance

No matter what conclusion we might draw from the study, the outcomes from this project will be valuable to other universities in similar situations. We believe that the results will enlighten us about student exchanges in computer science, about Chinese students in computer science (whether they are in China or elsewhere) and about internationalisation of Chinese universities. It will also offer insights about what we can gain, and what problems arise, when we teach computer science in an internationalised environment.

In order to cope with the internationalised curriculum and mixed Chinese/Western classrooms, students have to change their way of learning. They need to learn to work with people with different cultural backgrounds and learn from one another in new ways. Both incoming and outgoing students experience different learning

environments<sup>1</sup> and bring changes to the existing systems at the same time. The teachers might need to refine their teaching strategies, but must still meet the demands of the local students.

Further, we assume that an investigation of the learning and teaching of computer science from the perspective of Chinese students and teachers in an international education context will enhance our understanding of the nature of learning, both of Chinese and non-Chinese students. It will thus provide insights that can be used to improve the development of learning and teaching.

### 3 Related Research

As this project spans a broad area, we have structured the section concerning related research according to three major themes: internationalisation at an institutional level, Chinese students, Computer Science Education.

#### 3.1 Internationalisation at an institutional level

In general, the education and learning traditions of China are heavily influenced by its Confucian heritage (Bush & Qiang, 2000). Confucius's definition of the purpose of education as changing people for the better remains at the heart. Although modern schooling has been accompanied by attitudinal changes, "the Chinese people have not lightly discarded the patterns of thinking and action from their rich historical past whose values have permeated the new Marxist precepts" (Cleverley, 1991, p.xii).

To some extent, Chinese academics are still relatively isolated from current developments in Western countries. Postiglione and Mingle (1999) report that although 95% of 276 academics from three universities in Shanghai agree that international connections are important, only 11% of them have studied or travelled overseas in the past three years. There is also general concern about the quality of Chinese higher education due to insufficient funding and poor infrastructure (Postiglione & Mingle, 1999).

In the context of internationalisation, institutions of higher education in China are looking to benefit from "new skills and different approaches brought by international scholars, better international research networks, or enhanced communication skills" (Welch, 1997, p.336). One of the means to reach these aims is through joint programs with foreign universities, which have increased rapidly since the 1990s. These programs have played an important role in facilitating human resource development and capacity, making use of foreign educational resources in the globalised world (The National Centre for Educational Development Research, 2002).

In recent years, the number of incoming international students in Chinese higher education institutions has grown dramatically, but compared to many Western universities, the degree of internationalisation is still low.

International students have moved from courses specifically designated for them to places where they can take the same courses as Chinese students. There is also an increase in the number of non-degree programs delivered in English for short-term international students and a growth in the number of degree-conferring programs in English at both undergraduate and postgraduate levels (Huang, 2006). Internationalisation is also taking place in the part of the curriculum that is intended mainly for domestic students. By the end of 2006, more than 100 higher education institutions and about 165 joint programs in China have been approved by the authorities<sup>2</sup>.

This summary shows that much research has been done on internationalisation from an organisational perspective. Still, the teachers' and students' own experiences of internationalisation in their educational context have until now been under-investigated (Wihlborg, 2005).

#### 3.2 Chinese students

Previous research on Chinese learners, published in English, can broadly be characterised into two categories: The first category investigates how Chinese learners living in Asia learn. These studies often reveal and focus on factors such as individual/collective orientations, socialisation, or forms of motivation, with the aim of offering a nuanced picture of the Chinese learner (see for example Watkins & Biggs (1996, 2001) and references in these books). They frequently draw comparisons with the Western situation, rather than studying Chinese learners in their own right. The second category contains projects that explore the situation of Asian learners who live and study in Western countries. These projects generally see these learners as a minority with its own particular problems and strengths (see for example Volet (1999)). Again, the Western values come to serve as a frame of reference.

#### 3.3 Computer Science Education

Much effort has been put into exploring how students learn computer science. Although international aspects have been discussed frequently within the computer science education community, they are only rarely studied in empirically based projects. The Runestone initiative (Last, Almstrum, Daniels, Erickson & Klein, 2000), in which students in two different countries jointly develop an advanced software system, is one of few exceptions. The students' learning of computer science, their collaboration and their experience of the situation have been investigated by Hause & Woodroffe (2001) and by the second author of this paper, Anders Berglund (2005). Of particular interest in the context of the proposed project is that Berglund has used phenomenography (Marton & Booth, 1997) as his principal research framework. This approach has been used fairly frequently in computer science education research; see Berglund (2006) for an overview.

<sup>1</sup> We use the term *learning environment* in a broad sense in this paper. It encompasses many components, such as teaching, literature, atmosphere, and, of course, the physical environment.

<sup>2</sup> <http://www.jsj.edu.cn/mingdan/002.html>, accessed on August 17, 2007

Our literature overview has not revealed any studies related to student exchanges in computer science that discuss the learning of computer science or the students' experiences of their learning. Further, we have not found any studies exploring the learning of computer science from the point of view of Chinese learners.

## 4 The Upcoming Project

### 4.1 The Setting

The School of Software Engineering (SSE) in Tongji University, Shanghai, China, is a newly established school, founded in 2002. Due to its specialty in the IT industry, internationalisation is an important aim for the school. The school has currently several types of international program: student exchange programs, staff exchange programs, international internship programs and joint-degree programs. In order to recruit international students, the school offers many courses in English and uses English textbooks in most courses. This greatly improves students' ability to use English and facilitates communication in the international context.

More and more international students study in the school and join courses that were originally intended only for Chinese students. These international students come from different countries and are in different socio-economic situations. They bring not only different ways of studying and thinking about computer science, but also different cultures and values. Chinese students and staff have to learn how to cope with these students and might need to change their own way of acting. This is the inspiration of the proposed research project, which will explore the ways and the degree to which internationalisation influences the students' learning and the teachers' work in SSE.

### 4.2 Research Questions

With this as a background, the following research questions have been proposed for the project:

- 1) How does internationalisation affect learning in computer science in SSE?
  - a) How does internationalisation affect the learning environment in SSE?
  - b) How do Chinese students and foreign students affect each other's ways of study?
- 2) How does internationalisation influence teaching in SSE?
  - a) How does internationalisation influence teaching strategies?
  - b) How does the internationalisation of the curriculum affect the quality of teaching?

### 4.3 Methodology

Based on Crotty's (1998) work, the methodological framework for this study is:

- Epistemology – Constructionism
- Theoretical Perspective – Interpretivism

- Methodology – Phenomenography
- Methods – Interview

The outcome of the project will be descriptive, and will be based mainly in qualitative research approaches, particularly phenomenography (Marton & Booth, 1997).

### 4.4 Phenomenography

Phenomenographic studies strive to discover and describe the different ways in which people understand or experience certain phenomena (Marton & Booth, 1997). The common method to collect data for phenomenographic studies is through open, deep interviews (Booth, 1997). *Open* indicates that there is no definite structure to the interview, while *deep* indicates that the interview will follow a certain line of questioning until it is exhausted, until the participant has nothing else to say or until the researcher and participant have reached some kind of common understanding about the topics of discussion.

### 4.5 Data Collection

In a phenomenographic research project, the interviewees are typically asked to respond to open-ended questions about the particular phenomenon being studied. Their responses are analysed into conceptual categories on the basis of qualitative similarities and differences.

The students will be chosen for interviews based on a sampling design, such as incoming exchange students, outgoing exchange students/interns and regular students who do not attend any international program. The questions will be designed so that the interviews can reveal differences in the students' experiences of learning and of the different cultures in their learning environment.

All teachers at SSE will be invited to take part in the study. Therefore the individual interview, aiming to collect teachers' experiences of their teaching, will be complemented with Russell's synergetic focus group discussions (Russell, 1994). Focus groups are distinguished from group interviews because they explicitly include participant interaction and they seek to be more than mere conversations. Therefore, analysts can find more influencing factors (Wilson, 1997).

Students and staff from academic partners, such as the Department of Information Technology, Uppsala University, Sweden, and some personnel from partner companies will be included in the study.

### 4.6 Analysis

Each interview will be recorded, transcribed verbatim and analyzed following the principles of phenomenographic analysis. The data analysis process will involve reading of transcripts and progressive refining of emerging categories of description. The data will be pooled and an iterative approach will be adopted to ascertain relations, similarities and differences in the responses. The fundamental concern is to understand the meanings of learning and teaching inherent in the responses.

## 5 Discussion Topics

This paper started by asking two questions. These questions can be slightly reformulated to serve as discussion topics:

- In which ways does internationalisation affect how our students learn computer science? This is, as we have discussed in this paper, one of the main research questions in this project, and the outcome of the project is expected to inform us on this. Still, as the project is to a large degree explorative, in that it aims to address a domain that has previously been little researched, speculations about the expected outcome can support the work, for example through a more focused data collection.
- What can the university learn about teaching of computer science from international collaborations?

We also wish to discuss a third question:

- The project is theoretically rooted in phenomenography. Would this approach be sufficient for tackling these complex research questions, or are complementary perspectives needed?

## 6 References

- Berglund, A. (2005): *Learning computer systems in a distributed project course: The what, why, how and where* (Vol. 62). Uppsala, Sweden: Acta Universitatis Upsaliensis.
- Berglund, A. (2006): Phenomenography as a way to research learning in computing. *Bulletin of the National Advisory Committee on Computing Qualifications, BACIT*, 4(1).
- Booth, S. (1997): On Phenomenography, Learning and Teaching. *Higher Education Research and Development*, 16, 135-159.
- Bush, T. and Qiang, H. (2000): Leadership and Culture in Chinese Education. *Asia Pacific Journal of Education*, 20(2), 58-67.
- Cleverley, J. (1991): *The Schooling of China: Tradition and Modernity in Chinese Education* (2nd ed.). Sydney, Australia: Allen & Unwin.
- Crotty, M. (1998): *The Foundations of Social Research: Meaning and Perspective in the Research Process*. Crows Nest, NSW, Australia: Allan & Unwin.
- Hause, M. and Woodroffe, M. (2001): *Team Performance Factors in Distributed Collaborative Software Development*. Paper presented at the *Proc. Psychology of Programmers Interest Group (PPIG)*, Bournemouth, U.K.
- Huang, F. (2006): Internationalization of curricula in higher education institutions in comparative perspectives: Case studies of China, Japan and The Netherlands. *Higher Education* 51, 521-539.
- Last, M., Almstrum, V., Daniels, M., Erickson, C. and Klein, B. (2000): An International Student/Faculty Collaboration: The Runestone Project. *ACM SIGCSE Bulletin*, 32(3), 128 - 131.
- Volume 32 , Issue 3 Sept. 2000
- Marton, F. and Booth, S. (1997): *Learning and Awareness*, Mahwah, NJ, USA: Lawrence Erlbaum Associates.
- Postiglione, G. A. and Mingle, J. (1999): Academic Culture in Shanghai's Universities. *International Higher Education*, 17, 12-13.
- Russell, A. (1994): *Synergetic Focus Groups: A Data Gathering Method for Phenomenographic Research*, School of Language and Literacy Education, Queensland University of Technology, Brisbane, Australia.
- The National Center for Educational Development Research, P. R. C. (2002): *Green Paper on Education in China: Annual Report on Policies of China's Education*. Beijing: Jiaoyu kexue chubanshe [Educational Science Press]. [In Chinese]
- Volet, S. (1999): Learning across cultures: appropriateness of knowledge transfer. *International Journal of Educational Research*, 31:625 - 643.
- Watkins, D. and Biggs, J. (1996): *The Chinese Learner: Cultural, Psychological, and Contextual Influences*. Hong Kong: Comparative Education Research Centre, University of Hong Kong.
- Watkins, D. and Biggs, J. (2001): *Teaching the Chinese Learner: Psychological and Pedagogical Perspectives*. Hong Kong: Comparative Educational Research Centre, University of Hong Kong.
- Welch, A. (1997): The Peripatetic Professor: The Internationalisation of the Academic Profession. *Higher Education*, 34(3), 323-345.
- Wihlborg, M. (2005): *A pedagogical stance on internationalizing education: An empirical study of Swedish nurse education from the perspectives of students and teachers*. Lund University, Lund, Sweden.
- Wilson, V. (1997): Focus groups: a useful qualitative method for educational research. *British Educational Research Journal*, 23(2), 209-224.

# DEMO/POSTER PAPERS



# Why should we bore students when teaching CS?

Tuukka Ahoniemi

Essi Lahtinen

Keeko Valaskala

Institute of Software Systems  
Tampere University of Technology,  
PO Box 553, Tampere, Finland  
{tuukka.ahoniemi, essi.lahtinen}@tut.fi

## Abstract

Motivating students is extremely important. This poster is about pursuing the use of humour, or at least real-life related examples, on assignments in CS courses. We conducted the wild experiment of disguising a boring assignment description as a humorous pirate story. The outcome was surprisingly positive. This sort of approach can have a direct and positive effect on the dropout rate of the course, so it is worth trying.

## 1 Motivation for Motivation

The skills that students are supposed to learn while doing their programming assignments are complex and multiple. This leads to long and boring assignment descriptions that are not interesting enough to inspire careful reading. Thus the students find the whole assignment unmotivating and also pose questions that are already answered in the assignment description. This certainly does not help with the high dropout rates in programming courses.

Our discipline is suffering from a loss of applicants (Curzon 2007), though the employment rates are staying high (at least in Finland). This indicates that the field has become unattractive to some extent (Klawe 2005). If the assignment descriptions in the first-year courses are all complex, boring, or involving mathematical formulas, the students do not relate our teaching to the ongoing high-drive ICT hype (Web 2.0 etc). So the problem can be to some (rather tiny) extent generalized to be a matter of the attractiveness of the whole discipline: are we boring or not?

## 2 The Course Atmosphere

Our approach to maintaining the students' interest and motivation towards the course is to create a unique and friendly atmosphere in the course. This does not simply mean being friendly and smiling when lecturing. It requires the atmosphere to be integrated to the whole course as many students do not attend lectures, and the most demanding part of the course is what is to be done at home – the assignments. Those are the parts that need to reflect the attitude of the instructors instead of demotivating the student. That is why we suggest making the assignment descriptions motivating and fun.

The actual core of the assignment can be designed to match its pedagogical purpose, but our point is

how it is presented. Plain percentage calculation might be a thriller for a scientist, but calculating your mid-term party punch alcohol percentages could be more interesting for a student. The idea is not only to show something of a real-life example but to actively use humour. Matocha et al. (1998) have named this sort of unorthodox teaching method *extended analogy*. A similar approach with the same underlying principles is used effectively by Dr. Paul Curzon in doing CS outreach for kids (Curzon 2007).

## 3 A Part of Naval War-history – A Case Study in TUT

The introductory programming course in Tampere University of Technology includes a relatively large programming assignment which has been a demotivator for most of the students. To fight the boredom and the huge dropout rate we decided to disguise our well designed learning objectives which ultimately entailed a long assignment description. The actual assignment was to implement a simple text-based version of the classic Battleship game; this was hardly boring, but not exactly hilarious either.

In a short meeting we decided to come up with a frame story. We wrote an introductory chapter full of rubbish to the assignment description. Following quite closely the plot of the blockbuster film 'Pirates of the Caribbean – Dead Man's Chest' by Depp et al. (Verbinski 2006) we turned our Battleship game into 'Pirates of Kaukajärvi<sup>1</sup> – (Ph)Doctor Man's Chest'. The example execution printouts in the assignment description were also turned humorous, resulting in quite a package.

Besides disguising the assignment description, the course staff, a subset of the authors<sup>2</sup>, decided to stick their necks out when presenting the description to the students in the lecture. The result was a one-hour spectacle of two pirates lecturing about the programming project, with numerous laughs (see Figure 1).

The experiment had no ambitious goals, besides amusing the lecturers themselves, so in collecting feedback from the assignment in general, as we always do, we were overwhelmed by the amount of extra positive feedback relating to the frame story – and we had not even asked students explicitly to comment on that. Indeed, we received not a single negative comment related to the frame story. Here are some extracts from the student feedback:

*"It's nice that school doesn't always have to be too formal and unhumorous. Despite the actual programming would be as boring as always it's cheering that the subject is nice."*

<sup>1</sup>Kaukajärvi is a lake near our university.

<sup>2</sup>Keeko is not a teacher, but an imaginary polar adventurer presenting here a meta-example of the subject.



Figure 1: Sticking one's neck out for an educational purpose

*"The subject of the assignment was BRILLIANT! The subject was the biggest reason I kept motivated even during the most laborious moments."*

*"A student in another university gave me bitter feedback on how they can never do anything this nice but always have to be so serious."*

*"The introduction in the assignment description gave me the best laughs in the semester, thank you!"*

#### 4 Discussion

Disguising assignment descriptions with hilarious details or relating them to real-life examples does not require too much additional work. However, the outcome can be extremely positive as students feel a unique course atmosphere. The use of comedy is a huge aid in creating the atmosphere; but of course it is not natural for all instructors and should not be forced. Real-life related, non-humorous examples and assignments are still more motivating than purely theoretical ones.

An additional benefit of using humour is showing the students that despite his/her additional knowledge, the lecturer is still a human being and can be approached for conversation and questions. The freshmen especially tend to find the status of a lecturer so frightening that they might not ask for help even when they really need it. This could already be helpful in decreasing the dropout rate.

In some cultures, however, and in lower levels of school, it can be really important for the lecturer to maintain the status of ultimate authority, so there can be some sort of moral conflict between maintaining the authority and trying to be approachable, especially through the use of humour.

Our experience shows that the broad use of humour and well designed real-life related examples and assignments throughout the course has created a suc-

cessful course atmosphere where the students dare to talk not only with the teaching assistants but also in the lectures (in a positive way). The students liked the course atmosphere, and even the instructors enjoyed their work much more.

#### 5 Acknowledgments

The Nokia Foundation has funded part of this work.

#### References

- Curzon, P. (2007), Serious fun in computer science, in 'ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 1–1.
- Klawe, M. (2005), Changing the image of computer science: a north american perspective in conversation with europe, in 'ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education', ACM Press, New York, NY, USA, pp. 3–3.
- Matocha, J., Camp, T. & Hooper, R. (1998), Extended analogy: an alternative lecture method, in 'SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education', ACM, New York, NY, USA, pp. 262–266.
- Verbinski, G. (2006), 'Pirates of the Caribbean – The Motion Picture. The Official Website', <http://disney.go.com/disneypictures/pirates/>.

# Nalkki-project – Tool for Plagiarism Detection Using the Web

Petri Sirkkala

Sami Puonti

Institute of Software Systems  
Tampere University of Technology,  
P.O.Box 553, FI-33101 Tampere, Finland,  
Email: {petri.sirkkala, sami.puonti}@tut.fi

## Abstract

Students achieve the best results in learning by writing and doing exercises. This mandates a large number of written exercises. However, limited resources and distribution of assessment work lead to problems when students' answers need to be checked for plagiarism. Plagiarism or copy pasting is difficult to notice in a large volume of documents. The demonstrated project focuses on computer-assisted plagiarism detection in medium to large volumes of text-based submissions. Moreover, the project supports automated search for web sources.

**Keywords:** Cheating, plagiarism, computer assisted assessment, text documents, web based

## 1 Introduction

A growing number of students who take programming courses in Tampere University of Technology use the Internet as a source for study material. Also many advanced courses point to the Internet for more up-to-date information on the subject of the course. It is estimated in Jones (2002) that nearly three-quarters of students in college use the Internet more than the library. As a result, the studied materials are easier to copy into exercise submissions. It was indicated by Culwin et al. (2001) that cases of plagiarism in initial programming courses were evident and currently less than 20% of the cohort. However, they added that plagiarism was clearly increasing. Students on a large mass course can easily produce hundreds of submitted text works. To save time, plagiarism detection needs to be automated as far as is reasonable and reliable.

Nalkki is a web application that helps to find plagiarism in students' submissions. Nalkki consists of a core and a web user interface that are written in Python. It can be installed locally in the university or department that wishes to use it. As the deployment is local it is easier to maintain security and to limit access to students' answers.

Automatic comparison is only the first step in determining if the submission really is plagiarised, and manual work is always required to make a decision on the findings (Surakka et al. 2006). For making this decision Nalkki offers two views. First it shows a listing of all documents ordered by relative similarity. After finding an interesting case in the similarity listing, the teacher opens a view to see the detailed comparison findings.

The demonstrated project is based on a Virtual university funded project, which was further enhanced with web aware features.

## 2 Problems in finding plagiarised text

Finding plagiarised parts of a text document is very slow labour for teachers. Even with a limited number of texts it relies on the teacher's ability to read and remember every submission. This is slow and ineffective. The Internet makes the problem even worse, since the teacher would be required to read and remember related material on the Internet to find the possible sources of copy and paste. Search engines can certainly help to find sources that have been copied, but picking suitable search terms and manually searching is tedious and repetitive.

As the process of finding plagiarised parts is based on the teacher's ability to remember all that he or she has read, the results may be incomplete. Some clear cases of copy and paste may easily be overlooked. And since the workload cannot be shared between multiple assistants, the monolithic toil is easily impossible for one teacher.

Plagiarism eats resources that would be better used in real work. It is wasting everyone's time without any gain on the course material. On mass courses, some students always play foul game. Without proper tools to handle plagiarism, the chances that those students will be penalised are close to nothing. We present Nalkki, a tool to help detect plagiarism.

## 3 Automated plagiarism search process

The process of finding plagiarism with Nalkki is generally done in four wizard-style steps. Adding submissions to Nalkki is the first step. Submissions are uploaded to Nalkki as individual files or as a zip file containing the source documents. The second step is running the comparison process. The comparison includes converting documents into a suitable text format, scanning for given references and making search queries to find sources that are not directly referenced. The third step is viewing the results in a listing. The last step is checking the findings in a detailed comparison view of each potential document.

### 3.1 Presenting the similarity listing

Nalkki presents two views of the results. The first view is a similarity listing of all source documents, sorted in relative similarity order. By relative similarity we mean that the number of words that are found to be copied from another source is divided by the length of the document in words. This listing allows the comparison view to be opened.

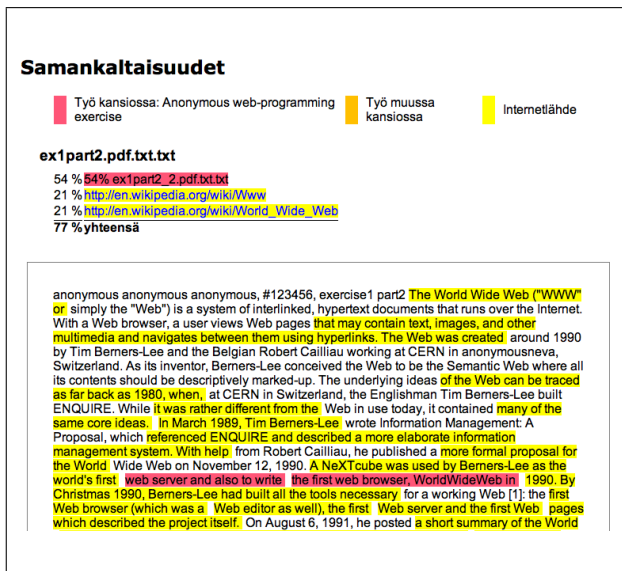


Figure 1: Comparison view reveals details about copied content

In the similarity listing the first column tells how much of the whole submission is found to have matches in other documents. The second column shows what was the biggest single match between this submission and another document. The third column leads to the comparison view that is explained in the next section. In the fourth column is the original name of the submission.

### 3.2 Viewing detailed comparison of a document

Any document that is listed in the similarity view has a link to open the document for closer inspection. This link leads to the comparison view, where the document text is presented. Any copied parts are identified by colours. Figure 1 shows an example of the comparison view.

The comparison view uses three colours to highlight the matches Nalkki has found. Red highlight is used to denote a match that was found only in another document in this submission set. Yellow means that the match was found on the Internet. If a match leads both to another document in this submission set and to the Internet, the Internet colour dominates as it is assumed that both submissions have copied the same Internet source. An orange highlight is used if the match is made with a submission in a previous submission set, for example last year's submission of the same subject.

## 4 Discussion

Automated plagiarism detection works well when enough comparison material is available. When the exercise is short or there are few participants the results are not so good. Fortunately, these cases are better handled by manual inspection. As noted by Ahoniemi & Reinikainen (2006), some assignments tend to produce similar results. For example, a teacher might give students a base document to work on. The text in the base document would therefore be present in most of the submissions. The unwanted comparison matches could be compensated by letting the teacher upload a negative match document that could be used to eliminate those matches.

The conversion to text format is limited to certain document types. This limits Nalkki in two ways.

First, the submissions are required to use formats that are suitable for text conversion. Second, external web documents that Nalkki cannot convert to text are not used in comparison. It might be possible to support more formats for web documents by allowing partial failure in conversion. Currently Nalkki supports only essay-style answers, and is therefore not directly usable for finding plagiarism in program code.

Nalkki only provides clues to potential cases of plagiarism. It is necessary to refer to the original document to prove the case. As stated by Surakka et al. (2006), finding potential plagiarism cases takes only 10-40% of the time, while an estimated 60-90% of the time is spent on manual work after detection. Nalkki has proven to be effective in the courses it has been piloted on. Some cases of plagiarism have already been detected with Nalkki. The feedback from teachers has been positive. Moreover we have been asked to present the system at faculties' teacher meetings. Just because Nalkki exists the teachers have already warned students not to try plagiarism, as the chances of getting caught are high.

Since plagiarism is a delicate subject and there are legal limitations to sharing information, the reports are not generally available. Plagiarism is a very serious accusation and therefore each case must always be double-checked by hand.

## 5 Conclusions

Plagiarism is a growing issue. To manage the issue, automated detection tools help to find cases of copy and paste more efficiently in medium to large volumes of submissions. Teachers have found Nalkki to be a useful tool in detecting plagiarism and used it as a threat to prevent plagiarism. Since plagiarism is such a serious accusation, the final decision must always be made manually.

## References

- Ahoniemi, T. & Reinikainen, T. (2006), Aloha - a grading tool for semi-automatic assessment of mass programming courses, in 'A. Berglund & M. Wiggberg (Eds.) Proceedings of the 6th Baltic Sea Conference on Computing Education Research', Uppsala, Sweden, pp. 139-140.  
URL: <http://cs.joensuu.fi/kolistelut/>
- Culwin, F., MacLeod, A. & Lancaster, T. (2001), 'Source Code Plagiarism in UK HE Computing Schools, Issues, Attitudes and Tools', *South Bank University Technical Report SBUCISM-01-02*.
- Jones, S. (2002), 'The Internet goes to college', *Pew Internet & American Life* 15.
- Surakka, S., Ahtiainen, A. & Rahikainen, M. (2006), Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises, in 'A. Berglund & M. Wiggberg (Eds.) Proceedings of the 6th Baltic Sea Conference on Computing Education Research', Uppsala University, Uppsala, Sweden, pp. 141-143.  
URL: <http://cs.joensuu.fi/kolistelut/>

## Author Index

- Ahoniemi, Tuukka, 163, 195, 227
- Back, Ralph-Johan, 167  
 Bednarik, Roman, 203  
 Bennedsen, Jens, 21  
 Berglund, Anders, 77, 171, 175, 207, 221  
 Beynon, Meurig, 31  
 Brabrand, Claus, 3
- Carbone, Angela, 109
- Dahl, Bettina, 3  
 Davies, Janet, 217  
 de Raadt, Michael, 41, 53
- East, Philip, 207  
 Eckerdal, Anna, 207  
 Eremin, Evgeny A, 179  
 Erkkola, Teemu, 163
- Freischlad, Stefan, 183  
 Fuller, Ursula, 187
- Harfield, Anthony, 31
- Islas Sedano, Carolina, 213
- Järvinen, Hannu-Matti, 199
- Kaila, Erkki, 151  
 Keim, Bob, 187  
 Kinnunen, Päivi, 175, 207  
 Kinshuk, , 217  
 Knobelsdorf, Maria, 65  
 Kohl, Lutz, 191  
 Kuts, Ekaterina, 213
- Laakso, Mikko-Jussi, 151  
 Lahtinen, Essi, 163, 195, 199, 227  
 Lai, David, 53  
 Lister, Raymond, iii, 171  
 Lönnberg, Jan, 77
- Luoma, Harri, 199
- Malmi, Lauri, 175, 207  
 Mannila, Linda, 167  
 McCartney, Robert, 207  
 Moreno, Andrés, 203  
 Moström, Jan-Erik, 207  
 Murphy, Laurie, 207  
 Myller, Niko, 203
- Pears, Arnold, 207  
 Peltomäki, Mia, 167  
 Puonti, Sami, 229
- Rajala, Teemu, 151  
 Ratcliffe, Mark Bartley, 207  
 Romeike, Ralf, 87
- Salakoski, Tapio, 151, 167  
 Salo, Anniina, 195  
 Schulte, Carsten, 21, 65, 207  
 Schwartzman, Leslie, 97  
 Sheard, Judy, 109  
 Simon, , iii, 119  
 Simon, Beth, 207  
 Sirkkala, Petri, 229  
 Sorva, Juha, 127  
 Stamouli, Ioanna, 207  
 Suhonen, Jarkko, 217  
 Sutinen, Erkki, 203, 213
- Thomas, Lynda, 207  
 Thompson, Errol, 217  
 Toleman, Mark, 41
- Valaskala, Keeko, 227  
 Vesisenaho, Mikko, 31
- Watson, Richard, 41, 53  
 Wiggberg, Mattias, 137
- Yang, Doris Dongsheng, 221

# Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

**Volume 67 - Conceptual Modelling 2007**

Edited by John F. Roddick, *Flinders University* and Annika Hinze, *University of Waikato, New Zealand*. January, 2007. 978-1-920682-48-4.

Contains the proceedings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM2007), Ballarat, Victoria, Australia, January 2007.

**Volume 68 - ACSW Frontiers 2007**

Edited by Ljiljana Brankovic, *University of Newcastle*, Paul Coddington, *University of Adelaide*, John F. Roddick, *Flinders University*, Chris Steketee, *University of South Australia*, Jim Warren, *the University of Auckland*, and Andrew Wendelborn, *University of Adelaide*. January, 2007. 978-1-920682-49-1.

Contains the proceedings of the ACSW Workshops - The Australasian Information Security Workshop: Privacy Enhancing Systems (AISW), the Australasian Symposium on Grid Computing and Research (AUSGRID), and the Australasian Workshop on Health Knowledge Management and Discovery (HKMD), Ballarat, Victoria, Australia, January 2007.

**Volume 69 - Safety Critical Systems and Software 2006**

Edited by Tony Cant, *Defence Science and Technology Organisation, Australia*. February, 2007. 978-1-920682-50-7.

Contains the proceedings of the 11th Australian Conference on Safety Critical Systems and Software, August 2006, Melbourne, Australia.

**Volume 70 - Data Mining and Analytics 2007**

Edited by Peter Christen, Paul Kennedy, Jiuyong Li, Inna Kolyshkina and Graham Williams. December, 2007. 978-1-920682-51-4.

Contains the proceedings of the 6th Australasian Data Mining Conference (AusDM 2007), Gold Coast, Australia. December 2007.

**Volume 72 - Advances in Ontologies 2006**

Edited by Mehmet Orgun, *Macquarie University* and Thomas Meyer, *National ICT Australia, Sydney*. December, 2006. 978-1-920682-53-8.

Contains the proceedings of the Australasian Ontology Workshop (AOW 2006), Hobart, Australia, December 2006.

**Volume 73 - Intelligent Systems for Bioinformatics 2006**

Edited by Mikael Boden and Timothy Bailey, *University of Queensland*. December, 2006. 978-1-920682-54-5.

Contains the proceedings of the AI 2006 Workshop on Intelligent Systems for Bioinformatics (WISB-2006), Hobart, Australia, December 2006.

**Volume 74 - Computer Science 2008**

Edited by Gillian Dobbie, *University of Auckland, New Zealand* and Bernard Mans, *Macquarie University*. January, 2008. 978-1-920682-55-2.

Contains the proceedings of the Thirty-First Australasian Computer Science Conference (ACSC2008), Wollongong, NSW, Australia, January 2008.

**Volume 75 - Database Technologies 2008**

Edited by Alan Fekete, *University of Sydney* and Xuemin Lin, *University of New South Wales*. January, 2008. 978-1-920682-56-9.

Contains the proceedings of the Nineteenth Australasian Database Conference (ADC2008), Wollongong, NSW, Australia, January 2008.

**Volume 76 - User Interfaces 2008**

Edited by Beryl Plimmer and Gerald Weber, *University of Auckland*. January, 2008. 978-1-920682-57-6.

Contains the proceedings of the Ninth Australasian User Interface Conference (AUI2008), Wollongong, NSW, Australia, January 2008.

**Volume 77 - Theory of Computing 2008**

Edited by James Harland, *RMIT University* and Prabhu Manyem, *University of Ballarat*. January, 2008. 978-1-920682-58-3.

Contains the proceedings of the Fourteenth Computing: The Australasian Theory Symposium (CATS2008), Wollongong, NSW, Australia, January 2008.

**Volume 78 - Computing Education 2008**

Edited by Simon, *University of Newcastle* and Margaret Hamilton, *RMIT University*. January, 2008. 978-1-920682-59-0.

Contains the proceedings of the Tenth Australasian Computing Education Conference (ACE2008), Wollongong, NSW, Australia, January 2008.

**Volume 79 - Conceptual Modelling 2008**

Edited by Annika Hinze, *University of Waikato, New Zealand* and Markus Kirchberg, *Massey University, New Zealand*. January, 2008. 978-1-920682-60-6.

Contains the proceedings of the Fifth Asia-Pacific Conference on Conceptual Modelling (APCCM2008), Wollongong, NSW, Australia, January 2008.

**Volume 80 - Health Data and Knowledge Management 2008**

Edited by James R. Warren, Ping Yu, John Yearwood and Jon D. Patrick. January, 2008. 978-1-920682-61-3.

Contains the proceedings of the Australasian Workshop on Health Data and Knowledge Management (HDKM 2008), Wollongong, NSW, Australia, January 2008.

**Volume 81 - Information Security 2008**

Edited by Ljiljana Brankovic, *University of Newcastle* and Mirka Miller, *University of Ballarat*. January, 2008. 978-1-920682-62-0.

Contains the proceedings of the Australasian Information Security Conference (AISC 2008), Wollongong, NSW, Australia, January 2008.

**Volume 82 - Grid Computing and e-Research**

Edited by Wayne Kelly and Paul Roe, *QUT*. January, 2008. 978-1-920682-63-7.

Contains the proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid 2008), Wollongong, NSW, Australia, January 2008.

**Volume 83 - Challenges in Conceptual Modelling**

Edited by John Grundy, *University of Auckland, New Zealand*, Sven Hartmann, *Massey University, New Zealand*, Alberto H.F. Laender, *UFMG, Brazil*, Leszek Maciaszek, *Macquarie University, Australia* and John F. Roddick, *Flinders University, Australia*. December, 2007. 978-1-920682-64-4.

Contains the tutorials, posters, panels and industrial contributions to the 26th International Conference on Conceptual Modeling - ER 2007.

**Volume 84 - Artificial Intelligence and Data Mining 2007**

Edited by Kok-Leong Ong, *Deakin University, Australia*, Wenyuan Li, *University of Texas at Dallas, USA* and Junbin Gao, *Charles Sturt University, Australia*. December, 2007. 978-1-920682-65-1.

Contains the proceedings of the 2nd International Workshop on Integrating AI and Data Mining (AIDM 2007), Gold Coast, Australia. December 2007.

**Volume 86 - Safety Critical Systems and Software 2007**

Edited by Tony Cant, *Defence Science and Technology Organisation, Australia*. December, 2007. 978-1-920682-67-5.

Contains the proceedings of the 12th Australian Conference on Safety Critical Systems and Software, August 2006, Adelaide, Australia.