UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 600

Novice Programming Students' Learning of Concepts and Practise

ANNA ECKERDAL

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009

ISSN 1651-6214
ISBN 978-91-554-7406-5
urn:nbn:se:uu:diva-9551

Dissertation presented at Uppsala University to be publicly examined in room 2446, Polacksbacken, Uppsala, Friday, March 6, 2009 at 10:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract
Eckerdal, A. 2009. Novice Programming Students' Learning of Concepts and Practise. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 600. 76 pp. Uppsala. ISBN 978-91-554-7406-5.

Computer programming is a core area in computer science education that involves practical as well as conceptual learning goals. The literature in programming education reports however that novice students have great problems in their learning. These problems apply to concepts as well as to practise.

The empirically based research presented in this thesis contributes to the body of knowledge on students' learning by investigating the relationship between conceptual and practical learning in novice student learning of programming. Previous research in programming education has focused either on students' practical or conceptual learning. The present research indicates however that students' problems with learning to program partly depend on a complex relationship and mutual dependence between the two.

The most significant finding is that practise, in terms of activities at different levels of proficiency, and qualitatively different conceptual understandings, have dimensions of variation in common.

An analytical model is suggested where the dimensions of variation relate both to concepts and activities. The implications of the model are several. With the dimensions of variation at the center of learning this implies that when students discern a dimension of variation, related conceptual understandings and the meaning embedded in related practises can be discerned.

Activities as well as concepts can relate to more than one dimension. Activities at a higher level of proficiency, as well as qualitatively richer understandings of concepts, relate to more dimensions of variation.

Concrete examples are given on how variation theory and patterns of variation can be applied in teaching programming. The results can be used by educators to help students discern dimensions of variation, and thus facilitate practical as well as conceptual learning.

Keywords: Computer science education, computer science education research, object-oriented programming, novice students, phenomenography, variation theory, dimensions of variation, learning, higher education, concepts, practise, Ways of Thinking and Practising

Anna Eckerdal, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1651-6214
ISBN 978-91-554-7406-5
urn:nbn:se:uu:diva-9551 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-9551)

To my children Per, Nils, and Olof

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Eckerdal, A., Thuné, M. (2005) Novice Java Progammers' Conceptions of "Object" and "Class", and Variation Theory. *SIGCSE Bulletin*, 37(3), pp.89–93

II Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Sanders, K., Zander, C. (2006) Putting Threshold Concepts into Context in Computer Science Education. *SIGCSE Bulletin*, 38(3), pp. 103–107

III Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliff, M., Sanders, K., Zander, C. (2007) Threshold Concepts in Computer Science: do they exist and are they useful? *SIGCSE Bulletin*, 39(1), pp. 504–508

IV Thuné, M., Eckerdal, A. (2009) Variation Theory Applied to Students' Conceptions of Computer Programming. *European Journal of Engineering Education*, Accepted for publication

V Eckerdal, A., Berglund, A. (2005) What does it take to learn 'programming thinking'? In Proceedings of the 1st International Computing Education Research Workshop, pp. 135–143.

VI McCartney, R., Eckerdal, A., Moström, J. E., Sanders, K., Zander, C. (2007) Successful students' strategies for getting unstuck. *SIGCSE Bulletin*, 39(3) pp. 156–160.

VII Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Zander, C. (2006) Categorizing Student Software Designs: Methods, results, and implications. *Computer Science Education*, 16(3), pp. 197–209.

VIII Eckerdal, A., McCartney, R., Moström, J., Sanders, K., Thomas, L., Zander, C. (2007) From *Limen* to *Lumen*: Computing students in liminal spaces. In Proceedings of the 3rd International Computing Education Research Workshop, pp. 123–132,

IX Eckerdal, A. (2009) Ways of Thinking and Pratising in Introductory Programming. Technical Report 2009-002, Department of Information Technology, Uppsala University, Sweden.

Comments on my contributions

In this section I list my main contributions for the papers included in this thesis.

I I planned and performed the data gathering and the writing, but in close discussion with the second author. Both authors separately analysed the data, and discussed the results until we came to an agreement.

II The six authors contributed equally to the background studies and writing of the paper.

III Data for this paper was gathered at three occasions. The first gathering was performed by four of the seven authors, me included. The second by two of the authors, not including me, and the last and most important and time consuming data gathering was planned and performed by all seven authors. The data analyses and writing were performed jointly by the seven authors of the paper.

IV I planned and performed the data gathering, but in close discussion with the first author. We analysed the data jointly. The first author outlined and wrote the paper, in discussion with me.

V I planned and performed the data gathering and the analysis, suggested the topic and wrote the paper, but in close discussion with the second author.

VI Data for this study was gathered by six researchers, where I was one. Five of the researchers, me included, performed the data analysis and writing jointly. The idea for the research came from one of the other authors.

VII Data for this paper was gathered by twentyone researchers. A subgroup of four researchers, me included, together with an additional researcher analysed a partial set of the data. The analysis and writing was jointly performed by the five authors of the paper.

VIII Data for this study was gathered by six researchers, where I was one. Five of these, plus a new member of the group are authors of the paper. I was the initiator of the paper. I suggested the topic, the analysis method, and outlined the structure of the paper. The data analysis the research builds on and the writing was performed jointly by the authors.

IX I am the sole author of this paper.

Contents

1 Introduction 11
1.1 Research questions 12
1.2 Terminology used in the thesis 13
1.3 Methodology 15
1.3.1 The first investigation 15
1.3.2 The second investigation 16
1.3.3 The third investigation 16
1.4 Overview of the thesis 16
1.4.1 Student learning of concepts 17
1.4.2 Student learning of practise 17
1.4.3 The relationship between conceptual and practical learning 18
2 The research in context 19
2.1 The Computer Science Education research field 19
2.2 The present research and the Computer Science Education research field 20
3 Research approaches 27
3.1 Qualitative research 27
3.2 Phenomenography 28
3.3 Content analysis 30
3.4 Trustworthiness in Qualitative Research 32
4 The present research 35
4.1 Research approaches applied in the present research 35
4.1.1 Phenomenography and variation theory in the present research 35
4.1.2 Content analysis in the present research 36
4.2 Trustworthiness of the present research 37
4.2.1 Trustworthiness in the first investigation 37
4.2.2 Trustworthiness in the second investigation 38
4.2.3 Trustworthiness in the third investigation 39
5 Results 41
5.1 Learning of concepts 41
5.2 Learning of practise 43
5.3 Ways of Thinking and Practising 45
6 Discussion - from a teaching perspective 49
6.1 Phenomenography in practise - an empirical example 49
6.1.1 The phenomenographic outcome space 50
6.1.2 Discernment and variation - identification of critical features 50

6.1.3 Dimensions of variation - open a space for learning 51
6.1.4 Implications for education - patterns of variation 52
6.1.5 The results related to previous research 55
6.2 Dimensions of variation and student learning of practise 57
7 Conclusions and future work 61
Summary in Swedish 65
Acknowledgements 68
Bibliography 69

1. Introduction

This thesis addresses the role of concepts and the role of practise in computer programming students' learning. Specifically, the relationship between practise and concepts in students' learning process is investigated. The research is empirically based on studies with students from several countries.

Computer science has conceptual[1] as well as practical learning goals (Roberts and Engel, 2001). Many computer programming students claim that "learning through practise" is by far the best way to learn to program, both regarding "learning to do practise" as regarding learning the concepts (Eckerdal, 2006). This claim seems to be supported by many educators, considering the huge amount of papers that have been published on how to help students to "learn through" and "learn to do" practise (Valentine, 2004; Gross and Powers, 2005). Still, after decades of attempts to improve the learning outcome (from an early reference from the childhood of programming education in 1969[2] it is reported about a "high withdrawal rate" from a course), mainly by focusing on the role of the practise, serious learning problems seem to prevail (Eckerdal et al., 2006b; Fleury, 2000; Fleury, 2001; Lister et al., 2004; McCracken et al., 2001; Robins et al., 2003). In contrast to this focus on practise, research in higher education in general has had a strong focus on students' conceptual learning (Entwistle, 2003; Entwistle, 2007; Molander, 1996; Molander et al., 2001; Posner et al., 1982).

The primary contribution of the present research lies in the investigation of the relationship between students' conceptual and practical learning, where the focus of the latter is on "learning to do" practise. First the role of practise and the role of concepts are investigated separately. It is shown that students have great difficulties in learning both concepts and practise. The investigations further strongly indicate complex relationships and mutual dependencies between practise and concepts in students' learning process. The results thus point to a need to further explore this relation. This has also previously been pointed to in science and technology education research (Séré, 2002; McCormick, 1997).

The thesis also contributes to the body of knowledge on learning in higher education by presenting an analytical model of how the learning of concepts and practise relate in novice programming students' learning.

The present research builds on three empirical investigations. Data from the investigations have been analysed from several different perspectives in order

[1] By conceptual learning goals I mean subject specific *concepts* students are supposed to learn, see Section 1.2.

[2] Retrieved 080812 from http://portalparts.acm.org/880000/873609/fm/frontmatter.pdf

to provide insights into students' learning of concepts as well as their learning of the practise.

There are two objectives of this thesis. The first is to contribute to the body of knowledge of the learning process in programming education. The second is pragmatic: to give concrete advice to programming educators on how teaching and learning can be improved.

1.1 Research questions

As a teacher in computer science my interest in students' learning led me to research on teaching and learning. A first investigation revealed novice students' understanding of some central concepts. At the same time the data showed the first traces of a complex relationship between practise and conceptual learning. One of the novice students in the study comments the course he or she had just finished, a first programming course:

> Yes, I think it has been difficult with concepts and stuff, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult [...] But I still think the course, it's difficult for a novice to sort of get a grip of how to study when you implement the programs and like that. (Eckerdal and Berglund, 2005, p. 138)

This student finds it difficult to learn the concepts, but what he or she emphasises is problems with "learning through practise", the same practise that is developed to help students learn the concepts. The novice students in this study specifically stressed the importance of practise in learning the concepts. But if "learning through practise" causes the largest problem, the students will neither learn "to do the practise", nor the concepts. Keeping in mind the substantial efforts in the last decades to improve "learning through practise" to facilitate novice students' learning (Valentine, 2004; Gross and Powers, 2005), it becomes obvious that practise is not merely the unproblematic road to conceptual learning.

The subsequently performed investigations established the important but problematic role of practise in programming education and led to my overarching research question:

- Which roles do practise and concepts play in programming students' learning process?

This question is broad, and to investigate all aspects of it is beyond the scope of a thesis. I have thus limited the question. To this end I have used a conceptual framework, Ways of Thinking and Practising, WTP (Entwistle, 2003; McCune and Hounsell, 2005), to embrace both the practical and the conceptual aspects of learning to program, and how these two are related.

The research questions highlighted in this thesis focuses on three themes related to the WTP framework. The first theme concerns students conceptual learning:

- How do novice programming students understand programming concepts?
- How do novice students understand what computer programming means, and how do they understand what *learning* to program means?
- How can the results from a phenomenographic outcome space inform teaching and thus improve learning in computer programming education?

The second theme concerns the role of practise in computer science education:

- What strategies do computer science students use when they are stuck in their learning?
- Can computer science students design software?

The third theme deals with how conceptual learning and practise are related in the learning process:

- How do students experience the process of learning threshold concepts, the so called *liminal space* (Meyer and Land, 2005) in computer science?
- How do practise and concepts relate in novice programming students' learning?

The first theme is discussed in Paper I through Paper V, the second in Paper VI and Paper VII, and the third theme is the topic of Paper VIII and Paper IX. The papers are separately described in Section 1.4.

1.2 Terminology used in the thesis

The focus of the thesis is to discuss the role of concepts and the role of practise in programming students' learning. My use of the word *concept* is broad. I will discuss concept as "an abstract or generic idea generalized from particular instances", and as "any idea of what a thing ought to be"[3]. Some of the concepts discussed in the thesis are lexical, word-sized, for example "class" and "object" while others are broader, for example "computer programming".

Practise is a broad term. In Paper IX I distinguish between skills, activities and exercises. Programming students are supposed to learn new practical skills like reading, writing, and debugging code. Each skill is manifested in many different activities that the students are supposed to learn, and these activities demand different levels of proficiency to be properly performed. For example the skill of reading code can, for a novice student, mean the recognition of key words in a program, while at a higher level of proficiency reading code implies being able to relate the code to a problem domain. Exercises on the other hand are here discussed in terms of practises where students follow more or less detailed instructions prepared by the teacher. Exercises are less discussed than the other two, since they represent "learning through practise", while the focus of this thesis is on "learning to do the practise".

Practise, in terms of exercises, is the main means to reach both conceptual and practical learning goals, for example reading, writing, and debugging

[3]Retrieved 090110 from Merriam-Webster Online Dictionary, http://www.merriam-webster.com/dictionary/

code. Computer programming thus involves "learning through the practise" as well as "learning to do the practise".

The rest of this section explains some computer specific terms used in the thesis. Other computing terms are defined when they are introduced in the text.

Computer science is a discipline in higher education which involves methods and theories underlying computers and software systems. *Software* comprises the computer programs, associated documentation and configuration data that is needed to make the programs work correctly. The purpose of producing software systems is to make computers solve problems.

The software development process is traditionally divided into several phases: problem analysis, software design, implementation and testing. Computer programming, which is sometimes used synonymously with implementation and sometimes in the broader sense as software development, is a core area in computer science. When *implementing*, the programmer writes *code* in a certain programming language, where code refers to the instructions which tell the computer what to do. These instructions follow rules from the particular programming language used. *Syntax* is the description of the possible combinations of symbols and specific words that are accepted in a programming language.

Software development involves use of certain software tools. For example, in the implementation and testing phase specific text *editors* are used that are developed to facilitate the implementation by, for example, recognising the syntax of the language. The editor and the compiler are often integrated in a development environment. The *compiler* is the software that translates (compiles) the code to a representation that is executable for a computer. When the code is tested it is checked to determine if it meets the requirements given. This involves *debugging* the code, which means finding and removing errors.

There exist several fundamentally different ways to tackle a problem for a program developer. Consequently there are different *programming paradigms* available. This thesis will discuss the object-oriented paradigm which is currently dominant in industry and university education. Examples of programming languages within the object-oriented paradigm are Java and C++.

Meyer (1988) describes the thoughts behind the *object oriented paradigm*:

> A software system is a set of mechanisms for performing certain actions on certain data. When laying out the architecture of a system, the software designer is confronted with a fundamental choice: should the structure be based on the actions or on the data? (Meyer, 1988, p. 41).

The latter choice is one of the main principles behind the object oriented paradigm. Meyer has the following definition of object-oriented design: "Object-oriented design is the method which leads to software architectures based on the objects every system or subsystem manipulates (rather than 'the' function it is meant to ensure)." (Meyer, 1988, p. 50).

The principal aim of software engineering is to produce programs with high quality, which is to say programs that are correct, efficient, reusable, extendible, easy to use, which are exactly the features that underpinned the development of the object-oriented paradigm (Meyer, 1988).

1.3 Methodology

My interest in understanding the student learning process, which appeared so difficult to penetrate, led me to investigate students' learning of concepts and practise, as presented in Section 1.1. The research aims to give a broad picture of students' learning experiences, emanating from the students' perspectives. Students experience learning as a whole, and in order to untangle the complex experience, several studies were performed. In this section this is described as three different investigations, although they together form the pool of empirical data that the research builds upon, and from which conclusions are drawn.

The data in the first investigation are interviews with novice programming students. The second investigation involves several data collections, including informal interviews and a questionnaire administrated to educators, and interviews with senior students. The third investigation includes a large set of data from senior students' doing a design task. In this way, data showing students' understanding of concepts as well as the role of practise in programming education were gathered.

1.3.1 The first investigation

The first investigation included in the present research is a study with 14 Swedish first year non-major computer science students. It is common at Swedish universities that non-major computer science students in technical and natural science education take at least one computing course where they are given an introduction to programming. The students had just finished their first programming course, using Java as the programming language. The aim of the investigation was to get a rich description of the variation in the students' different experiences of some concepts in object-oriented programming. The students were thus interviewed for example on their understanding of the concepts *object* and *class*, and what it means to learn to program. The answers to these questions were transcribed verbatim and translated to English where needed. The analysis was performed using a phenomenographic research approach, see Section 3.

The research questions informed by this study are how novice students understand what programming is and what learning to program means, how they understand central concepts in the object-oriented paradigm, and how the results from a phenomenographic outcome space can inform teaching. Furthermore data from the first investigation informed the question on how conceptual learning and practise relate in programming students' learning process.

1.3.2 The second investigation

The second investigation was performed by a group of researchers from Sweden, the United Kingdom, and the United States. The work was motivated by an interest in threshold concepts in computer science (Meyer and Land, 2005). Two pre-studies were performed with educators at two international conferences during the summer and fall of 2005. Educators were informally interviewed, and some answered a questionnaire. The aim was to find threshold concept candidates for further investigation. These two studies laid the foundation for an interview study with students, aiming at identifying threshold concepts from the students' perspectives. A subsequent multinational study with students from seven universities in the three countries was performed during spring 2006. 16 graduating computer science students were interviewed.

The interviews have been analysed from three different perspectives and inform the following research questions. The first analysis aimed at identifying threshold concepts in the discipline. The second analysed the parts of the interviews where the students discussed strategies for getting unstuck in their studies. The last analysis took a theoretical standpoint, aiming at investigating what *liminal space* means and involves in computer science. The theory of liminal space was used as a tool in the search for learning experiences characteristic of computer programming. Furthermore data from the second investigation informed the question on how conceptual learning and practise relate in programming students' learning process.

1.3.3 The third investigation

A multinational study was performed by 21 researchers at 21 institutions in the United States, the United Kingdom, Sweden, and New Zealand. This study involved 314 participants from three levels of education; students with low competence, graduating seniors, and educators (Tenenberg et al., 2005). The research presented in the present thesis was performed by a subgroup of the original 21 researchers, plus one researcher not participating in the original investigation. The data used for this research were software designs produced by a subset of the participants, the 149 near-graduation seniors. The participants were asked to design a "Super alarm clock" according to a number of criteria that were to be met. Beside these criteria, there was little guidance on how to perform the task. The designs were made on paper.

This investigation informs the research question on graduating computer science students' ability to design.

1.4 Overview of the thesis

As explained above, the papers in the thesis are organized around three themes. Concepts and practise are two inseparable and equally important learning goals in programming education. The themes thus focus on student

learning of concepts, student learning of practise, and the relationship between the two in students' learning process.

The first theme on student learning of concepts is discussed in the first five papers of the thesis.

The second theme, dealing with students' learning practise, is illuminated in Paper VI and Paper VII.

The last theme, how conceptual learning and learning practise are related in students' learning process is examined in Paper VIII and Paper IX.

1.4.1 Student learning of concepts

Student learning of concepts is researched at different levels of granularity. The analysis from a bird's eye view discusses students' understanding of what computer programming means, while the analysis at the next level aims at identifying central, threshold concepts in computer programming. Finally, the analysis that focuses primarily on details look at students' understanding of a few, possible threshold concepts. Below follows a description on the papers that belong to this theme.

At the most coarse-grained level Paper IV, *Variation Theory applied to Students' Conceptions of Computer Programming*, investigates students' understanding of the whole subject area, computer programming. This is followed in Paper V, *What does it take to learn 'programming thinking'?*, by an investigation of the same students' understanding of what *learning* computer programming means.

At the next level of granularity, Paper II, *Putting Threshold Concepts into Context in Computer Science Education*, identifies so called threshold concepts in computer science. Further Paper III, *Threshold Concepts in Computer Science: Do they exist and are they useful?*, investigates students' learning of such concepts.

Finally, at the most fine-grained level Paper I, *Novice Java Programmers' Conception of "Object" and "Class" and Variation Theory* presents an in-depth study of students' understanding of a few central concepts, concepts that are possible threshold concepts in object-oriented programming.

1.4.2 Student learning of practise

Learning computer programming concerns learning practical skills. In the present thesis Paper VII, *Categorizing Student Software Designs: Methods, results, and implications*, focuses on one specific skill, software design. Design is, beside writing code, reading code, and debugging code, considered as a core skill in programming education. The investigation on senior students' ability to design software is an important contribution to the body of knowledge of students' skillfulness.

Our investigation, together with related projects (McCracken et al., 2001; Whalley et al., 2006; Fitzgerald et al., 2008), all point to the conclusion that students have great problems in learning the practise. The learning outcome

in programming education has been argued to be closely related to good programming strategies (Robins et al., 2003; Davies, 1993). We have thus investigated such strategies in terms of what graduating students do when they are stuck in their learning. This line of research is presented in Paper VI, *Successful students' strategies for getting unstuck*. Some of the strategies identified and labeled in the paper have an abstract character, like "Be persistent/don't stop" or "See patterns". Others have a more concrete, practical nature, for example "Use a [software] tool", "Write programs" or "Trace [code]". Many of the strategies found in the analysis are thus related to the practical aspect of programming. In this way Paper VI emphasis the importance of students' learning practise and broadens the research presented in Paper VII which focuses on one particular aspect of practise, students' ability to design.

1.4.3 The relationship between conceptual and practical learning

The last theme presented in the thesis focuses on the complex relationship between conceptual and practical learning. The theme is highlighted by results from Paper I *Novice Java Programmers' Conception of "Object" and "Class" and Variation Theory*, Paper IV *Variation Theory applied to Students' Conceptions of Computer Programming*, and Paper V *What does it take to learn 'programming thinking'?* with novice students, but is established and elaborated in Paper VIII *From* Limen *to* Lumen*: Computing students in liminal spaces*, with senior students. The results of this analysis reveals a broad and rich picture of the students' learning experiences where the practise as well as the concepts play important but problematic roles in the students' learning process.

In this way Paper VIII gives a background for the analysis presented in Paper IX, *Ways of Thinking and Practising in Introductory Programming*. The paper is the synthesis of my thesis work. Important results from the first two investigations on conceptual and practical learning are discussed and further developed. The focus, discussed and analysed in depth, is however on how conceptual and practical learning relate in students' learning process.

2. The research in context

2.1 The Computer Science Education research field

Computer science[1] is a young discipline, only half a century old. As a discipline of its own, computer science education is even younger. Computer science has developed with an "astonishing pace" which has had "a profound effect on computer science education, affecting both content and pedagogy." (Roberts and Engel, 2001, Chapter 2) The rapid change of the subject matter taught has inevitably affected also the computer science education research discipline. The discipline has however encountered several problems. Berglund (2005) identifies some of them. First, the discipline is cross-disciplinary. It encompasses computer science, but in addition a range of other disciplines including pedagogy, psychology, learning technology, and more. According to Berglund, the lowest common denominator in this diverse field is "the *aim to improve learning and teaching within computer science,* and thereby to contribute to computer science." (Berglund, 2005, p. 23, italics in original) This is in line with the aim of the present research.

Berglund further points to the problem of knowing "who is 'in' the community." He writes, with reference to Clancy et al. (2001):

> As many of the leading researchers within the field are better known for their contribution to other sub-areas of computer science, it is also hard to determine where the edges of the community are. (Berglund, 2005, p. 23)

Another problem recognized, relevant for the present thesis, is that there has been, and still is a need of more qualitative research in computer science education research (Berglund et al., 2006). Berglund et al. (2006, p. 25) claim that "research into student learning is strengthened by increased awareness of the role and relevance of qualitative research approaches in CER."

A question that has been discussed in the CER community, and still is an issue, is how to define research in computer science education. What distinguishes research in teaching and learning from mere ideas of good teaching practise based on personal teaching experiences? This is debated for example in Goldweber et al. (2004), where one of the authors writes: "CSEd research is new. It

[1] Computer science is commonly abbreviated CS. Accordingly, computer science education is abbreviated CSEd or CSE, and computing education research CER.

co-exists in places with other sorts of publications (like SIGCSE) and where it starts and stops, where its edges are, are not yet clear."[2]

Fincher and Petre discuss how computer science education research has emerged as an "identifiable area" (Fincher and Petre, 2004, p. 1) during the past decades. The growth has come from different places like computer science practitioner conferences, sub-specialist areas like psychology of programming, and computer science research groups at different academic institutions. Another factor contributing to the shattered picture is the contributors, who have diverse expert knowledge like education, psychology, and different areas of computer science, and consequently have published in different research fora. Fincher and Petre write about this sprawling research field: "Despite this growth–and because of it–we are struggling to find the shape and culture of our literature." (p. 2)

Fincher and Petre discuss the characteristics of the publications that can be referred to as research: "they can be thought of as having two components: a dimension of rationale, argumentation or 'theory', and a dimension of empirical evidence." (p. 2)

The research presented in this thesis is well in line with the two criteria discussed by Fincher and Petre. All the papers build on empirical data (except Paper II, *Putting Threshold Concepts into Context in Computer Science Education*, which is a literature review) and they all include arguments, or theories, which the interpretations and inferences build on. Furthermore, all papers are published in well established fora, where the papers have been peer-reviewed by relevant specialists in computer science and/or education.

2.2 The present research and the Computer Science Education research field

Students' learning of computer science has been investigated from different perspectives. This section will put the present research in a context of research in computer science education, and specifically regarding research on students' learning computer programming which is a sub-field of the wider computer science education research field.

Pears et al. (2007) report on a literature survey on teaching of introductory programming. The following areas are investigated in the survey: Curricula, Pedagogy, Language choice, and Tools for teaching.

Randolph (2007) presents, from a positivistic perspective rooted in psychological research, a major overview of articles in computer science education. The author reviewed 352 computer science education articles published between 2000 and 2005. Randolph claims among other things that "several dif-

[2]SIGCSE mission statement, http://sigcse.org/about/, says: "The ACM Special Interest Group on Computer Science Education provides a forum for educators to discuss issues related to the development, implementation, and/or evaluation of computing programs, curricula, and courses, as well as syllabi, laboratories, and other elements of teaching and pedagogy."

ferences in research practises across the fields of computer science education, educational technology, and education research proper were found." (p. iv) Randolph furthermore found that one third of the articles reviewed "did not report research on human participants" and most of them "were program descriptions" (p. 173).

An older survey is by Austing et al. (1977) who report on literature in computer science education from the publication of the first ACM Computing Curricula 1968 (ACM Curriculum Committee on Computer Science, 1968), up to 1977, including for example survey reports, descriptions of programs, and descriptions of courses and other material.

A psychological/educational perspective on learning is the focus of Robins et al.'s review (2003) which compares "novice and expert programmers, programming knowledge and strategies, program generation and comprehension, and object-oriented versus procedural programming." (p. 137) Robins et al. specifically focus on "novice programming and topics relating to novice teaching and learning." (p. 137)

Simon (2007) summarises the range of different types of publications in computer science education. He presents an overview of classification of papers in the field that have been published in different fora. For example, Pears et al. (2005) present a classification which, with reference to Fincher and Petre (2004), suggests the following areas for computer science education *research*: studies in teaching, learning, and assessment; institutions and educational settings; problems and solutions; computing education research as a discipline.

The focus of the present research is on learning, namely programming students' learning of concepts and practise. Computer programming is one of the core areas in computer science education, which is established in the influential ACM/IEEE Computing Curricula 2001 (Roberts and Engel, 2001)[3]. Even though computer programming is a young discipline in higher education, students' difficulties are widely reported in the literature (Ben-Ari, 1998; Eckerdal and Thuné, 2005; Fleury, 1999; Fleury, 2000; Lister et al., 2004; McCracken et al., 2001; Robins et al., 2003).

The present thesis work is put in a research context below in the following way: first I present studies on students' conceptual understanding, which include questions on student understanding of single concepts as well as questions at a more coarse-grained level including student understanding of what computer programming is. Studies on student learning of practise is discussed from two perspectives. First, studies that investigate practise as a learning goal in terms of programming skills are presented. Then, studies investigating practise as a means to reach learning goals, conceptual as well as practical, are discussed. Because of the many published articles related to the psycholog-

[3]This curriculum is one in a series of curricula developed for computer science education, the first dating back to 1968 (ACM Curriculum Committee on Computer Science, 1968).

ical/educational study of programming, I will finally briefly touch upon this area of research, although it is not within the scope of the present thesis.

Student learning of concepts

As described in Section 1.4 the thesis presents research on student learning of concepts at different levels of granularity. At the most coarse-grained level are Paper IV, which discusses novice students' understanding of computer programming, and Paper V, which discusses the same students' understanding of what *learning* computer programming means.

Examples of research related to these questions are Booth (1992), who in her influential thesis investigates what it means and what it takes to learn to program, and Bruce et al. (2004) and Thuné and Eckerdal (2009), (Paper IV), who follow this line of research, studying students' understanding of what programming means. Similar research is presented by Eckerdal and Berglund (2005), (Paper V), and Stamouli and Huggard (2006) who investigate students' understanding of what learning to program means. The studies show very similar findings. Students' understandings vary from a narrow language-syntax-centered understanding to more desirable broader understandings including programming as problem solving, a skill that can be used outside of computing education.

Many studies point to the necessity of a good understanding of the central concepts within object-oriented programming. Ragonis and Ben-Ari (2005) present a long-term study on high school students' learning of concepts in object-oriented programming including "class vs. object, instantiation and constructors, simple vs. composed classes, and program flow. In total, 58 conceptions and difficulties were identified." (Ragonis and Ben-Ari, 2005, p. 203) Fleury (2000) found that students constructed their own understanding of concepts when they worked with programming assignments, and that those constructions were not always complete and correct. In a multinational study Sanders et al. (2008) investigated what novice object-oriented programming students see as the most important concepts, and how they express the relationships among those concepts. Some results from the study are that "[u]nlike earlier research, we found that our students generally connect classes with both data and behavior" (p. 332), but "few students see modeling as one of the most important OO concepts." (p. 336) Another multinational study, presented by Sanders et al (2005), involved 20 researchers and 276 participants from 20 different institutions. The study aimed to elicit novice object-oriented programmers' knowledge of programming concepts by using a "multiple, participant-defined, single-criterion card sort". The authors point to "the unexpected result that there were few discernible systematic differences in the population." (p. 121)

Examples of studies on students' conceptual understanding with a phenomenographic approach are Berglund (2005) who investigated senior students' understanding of concepts within computer systems, Boustedt (2007) who studied senior students' understanding of some advanced object-oriented

concepts, and Eckerdal and Thuné (2005), (Paper I), who investigated novice students' understanding of central object-oriented concepts.

Holmboe (1999) emphasises that good understanding in programming requires both practical skills and conceptual understanding, and a connection between the two. This mirrors the three foci of the present thesis. The following two sections will discuss the role of practise in computer science education.

Learning programming skills

There exists a considerable body of research on the role of practise in computer science education, both as means to reach the learning goals, and as a goal in itself. The latter is discussed in this section, in terms of skills students are supposed to learn.

A well known multinational study is McCracken et al. (2001) who investigated novice students' ability to write code. The authors concluded that many students can not program after their first introductory programming course, but lacked evidence for an explanation. Lister et al. (2004) continued the McCracken study and found that students' problems with programming "relate more to the ability of students to read code than to write it." (p. 139) This line of research has been extended by Whalley et al. (2006), who also study students' ability to read code. The authors found that "[s]tudents who cannot read a short piece of code and describe it in relational terms are not well equipped intellectually to write code of their own." (p. 251) In the same line of research is Lopez et al. (2008) who investigated the relationship between reading, tracing and writing code in novice students' learning. The authors found correlation between performance on "code tracing tasks" and "performance on code writing tasks" and also between "performance on 'explain in plain English' tasks and code writing." (p. 101)

Students' ability to debug code is investigated in a multinational study by Fitzgerald et al. (2008). The authors found that students that can debug are often good novice programmers, but the opposite does not always apply. On the other hand, "once students find bugs, they can fix them." (p. 93) Senior computer science students' ability to design software is investigated in a multinational study by Eckerdal et al. (2006), (Paper VII). The authors found that that only 9 % of the students produced partial or complete designs. We furthermore found that the number of academic courses taken by the students, the time the students spend on the design task, and the number of programming languages well known by the students were significantly correlated with the result of the design task. In summary, all the studies point to students' difficulties in learning the practical skills. This applies to novices as well as to senior students.

Practise as means for learning to program

Practise is often seen as an inevitable means to reach learning goals. Resources that enhance practise for learning are frequently discussed topics in conference papers and journal articles. Such a resource which is expected to have

high impact on object-oriented educations is the Java Task Force that was appointed by the ACM Education Board in 2004[4]. The mission was to develop a s collection of pedagogical resources that would support the use of Java in first-year computer science courses.

There is a strong focus on technology based learning support in the computer science education literature. This is pointed to by Valentine (2004) who did a meta-analysis on twenty years of proceedings from the largest conference in computer science education. The author categorized research papers dealing with beginning programming courses. During 1994-2003, 42% of the number of papers in the proceedings described software that was developed by the author of the paper to enhance learning.

Technology supported resources developed to enhance learning to program are discussed by Powers et al. (2006). According to the authors software resources developed to help novices to learn to program can be divided into several groups, for example *Narrative tools*, which "support programming to tell a story" and *Visual programming tools*, which "support the construction of programs through a drag-and drop interface". An example of the former is Alice (Powers et al., 2007) and an example of the latter is JPie (Goldman, 2004).

Ellis et al. (1998) report on technology supported resources for Problem Based Learning. For example, the authors discuss resources to provide subject guidance and information access, and resources to assist scaffolding. In the former group reference material like CD-ROM and the web is mentioned.

How do we know that technology based learning resources lead to good conceptual or practical learning? Gross and Powers (2005) performed an extensive literature search for assessments of the educational impact of novice programming environments. The authors relate their literature search to novice programmers learning difficulties saying that teachers "have developed a myriad of tools to help novices learn to program. Unfortunately, too little is known about the educational impact of these environments."

Pair programming has been greatly discussed in the computer science community during recent years. The fundamental thoughts behind pair programming are described as "students sit side-by-side at one computer to complete a task together, taking turns 'driving' and 'navigating.' " (VanDeGrift, 2004). Studies on the learning outcome of pair programming, and how pairs best are selected have been performed. Examples of this are VanDeGrift (2004) and Katira (2004).

Extreme programming (XP) has been discussed and used in industry, and to some extent in higher education. In XP planning, analyzing, and designing is done a little at a time, throughout software development. The XP practises also include other factors like pair programming and programmers' collective ownership of the code in the system (Beck and Andres, 2004).

[4]The reports from the Java Task Force with associated material are available from http://jtf.acm.org/index.html Retrieved November 18, 2008.

Other aspects of the practise discussed in the literature are the role of projects and programming assignments, and the roles of the programming language and programming environment. The former are discussed for example by Daly (2004) and Newman (2003). The latter are discussed in for example Kölling (1999a) and Kölling (1999b) where the author discusses where different programming languages and different programming environments are suitable.

Psychological/educational study of programming

As a contrast to my own research I will mention two large research areas in computer science education: research on students' misconceptions, and research comparing novices and experts behaviour. These research areas are close to the present research, but not the exact focus.

Robins et al. (2003) discuss "literature relating to the psychological/educational study of programming." The authors discuss "general trends", for example regarding comparison of novice and expert programmers. Examples from this line of research are Gugerty and Olson (1986) who compare expert and novice debuggers, Kahney (1983) who investigate novices' and experts' understanding of recursive procedures, Zou and Godfrey (2008) who investigate differences between newcomers' and experts' interaction with software development tools, and Winslow (1996) who, based on an overview of psychological research in programming pedagogy, claim that it takes 10 years of experience to turn a novice programmer into an expert.

Students' misconceptions are frequently reported in studies on students learning to program. I will mention a few. Ragonis and Ben-Ari (2005) present a large study with high school students learning to program. The article includes detailed lists of difficulties and misconceptions related to several concepts in object-oriented programming. Holland, Griffiths and Woodman (1997) claim that misconceptions of basic object concepts "can be hard to shift later. Such misconceptions can act as barriers through which later all teaching on the subject may be inadvertently filtered and distorted." Sanders and Thomas (2007) describe a close examination of student programs from an introductory programming course, in which they found evidence of misconceptions. Among other things they found difficulties in distinguishing between classes and objects, and in modelling.

Other programming paradigms are also discussed in this context. Spohrer and Soloway (1986) studied novice Pascal students and investigated "whether or not most novice programming bugs arise because students have misconceptions about the semantics of particular language constructs." (p. 183) The authors found that for most of the bugs investigated that was not the case. Bayman and Mayer (1983) report on a study on beginning BASIC programmers misconceptions of statements they had learned, and Fung et al. (1990) report on "novices' misconceptions about the interpreter in Prolog" (p. 311).

3. Research approaches

My research interest is in programming students' learning, and specifically in the students' own experiences of their learning. I first investigated students' experiences of some programming concepts, and how they went about learning the concepts. Aiming at describing this from the students' perspective the first two investigations mainly used interviews with students. Interviews with educators and a brief survey were initially used in the second investigation, but the results from initial analyses pointed to qualitative, student centered research methods, and consequently interviews with students were performed.

From the initial research questions, the data and the analyses led to new research questions concerning the role of practise in programming students' learning, but still related to how the students experience their learning.

The research questions presented in Section 1.1 suggest a predominantly qualitative research approach since the focus is on how-questions, see Section 3.1 below. Quantitative methods have been used to a minor extent, and only as a complement to a primary, qualitative approach.

In the present section I will briefly introduce the reader to qualitative research in general, and in particular to phenomenography and variation theory. Parts of the content analysis tradition will be discussed, namely qualitative content analysis, which has been applied in the present research. In addition, trustworthiness in qualitative research is introduced, and discussed in relation to the present work.

3.1 Qualitative research

Qualitative research is spread widely and cross cuts many disciplines, using a variety of methods and approaches. Denzin and Lincoln (2005) discuss the development of qualitative research. Considering its complex development, qualitative research is difficult to define. The authors still offer an "initial, generic definition" (p. 3):

> qualitative research involves an interpretive naturalistic approach to the world. This means that qualitative researchers study things in their natural settings, attempting to make sense of, or interpret, phenomena in terms of the meanings people bring to them. (Denzin and Lincoln, 2005, p. 3)[1]

[1] In the following text I will refer to this definition when I discuss the naturalistic paradigm.

According to Denzin and Lincoln (1994) qualitative researchers "seek answers to questions that stress how social experience is created and given meaning" in contrast to quantitative studies which "emphasize the measurement and analysis of causal relationships between variables, not processes." (Denzin and Lincoln, 1994, p. 4) Qualitative studies often focus on *Why?* and *How?* questions, less on *How much?* which is common in quantitative studies. The aim of qualitative research is rather to give "thick descriptions" of phenomena than to measure variables. To "make sense of" and look for "the meaning people bring" to phenomena are watchwords.

Examples of data collection methods used in qualitative research are participant observation and video recordings, but as Denzin and Lincoln (2005) write: "No specific method or practise can be privileged over any other." (p. 7). Empirical materials used involve for example interviews, artifacts, and historical texts "that describe routine and problematic moments and meanings in individuals' lives." (Denzin and Lincoln, 2005, p. 3-4)

The research questions (see Section 1.1) require analysis methods that can elicit the meaning embedded in the material. Content analysis is such a method that has been used on interviews and written artifacts to elicit meaning by categorisation. Phenomenography, here applied in the analyses of interviews, is another approach, used in educational research to understand differences in learning outcome by eliciting qualitatively different ways in which people experience phenomena.

3.2 Phenomenography

Phenomenography is a qualitative research approach, intended for educational research. Phenomenography was first developed in the 70's in Gothenburg, Sweden by a group of researchers. Ference Marton, Lars Owe Dahlgren, Lennart Svensson and Roger Säljö performed a study on students reading a text aiming at understanding the differences in students' understandings. They found clear qualitative variation in *what* the students understood, as well as *how* they went about studying the text. These findings have been used as a point of departure for research on learning in various subject areas in higher education, and have led to insights, such as the distinction between deep and surface approach to learning (Marton et al., 1984). From this empirical basis the phenomenographic research approach emerged, which focuses on describing and understanding the variation in how people experience phenomena in the world[2] *Phenomena* are described by Marton and Booth (1997) as the units that exceed a situation, bind it together with other situations and give it a meaning.

Marton and Booth (1997) write about variation in peoples' capabilities for experiencing the world:

[2]In the following text I will use *understanding* as interchangeable with *experience* since the present research discusses students' understandings of phenomena.

> These capabilities can, as a rule, be hierarchically ordered. Some capabilities can, from a point of view adopted in each case, be seen as more advanced, more complex, or more powerful than other capabilities. Differences between them are educationally critical differences, and changes between them we consider to be the most important kind of learning. (Marton and Booth, 1997, p. 111)

The object of interest in a phenomenographic study is thus how a certain phenomenon is experienced by a certain group of people, and the *variation* in the way the phenomenon is experienced (Marton and Booth, 1997, p. 110). It focuses on the students' perspectives and understandings, not on misconceptions. It does not take the researcher's perspective as the point of departure, but endeavours to adopt the student's perspective on learning. Marton and Svensson (1979) claim that in this perspective, the world as the student experiences it, becomes visible. The experience is a relation between the student and his or her world, it is not two independent descriptions, one of the student and one of the world. "[W]e have one description which is of a relational character." (Marton and Svensson, 1979, p. 472)

In phenomenographic studies, data are often gathered in the form of interviews where people are encouraged to describe their different experiences, or understandings of some phenomenon. The interviews are transcribed verbatim and the data, as text, are analysed. The analysis aims at identifying different understandings of the phenomenon discussed. The understandings are found when the data are read and reread and patterns of distinctly different understandings are looked for. Individual, decontextulised quotes illustrating certain understandings are compared with each other, grouped and regrouped, and eventually different categories of understanding emerge which form an outcome space. The quotes are also read and reread in their own context to make subtle distinctions to the researcher's understanding of the data. The researcher formulates the essence of the understandings found with his or her own words in the categories of description. In this iterative analysis, by again and again going back to the data, the categories of description finally emerge.

A fundamental assumption in phenomenography is that there exist only a limited number of qualitatively different ways in which a certain phenomenon can be understood. The categories in the outcome space show a "hierarchical structure of increasing complexity, inclusivity, or specificity" (Marton and Booth, 1997, p. 126). The categories describe the qualitatively different ways of experiencing the phenomenon that the researcher has identified in the data. Different categories reflect different combinations of features of the phenomenon which are present in the focal awareness at a particular point in time (Marton and Booth, 1997, p. 126). Marton, Runesson and Tsui (2004, p. 22) describe critical features: "the features that must be discerned in order to constitute the meaning aimed for."

The phenomenographic analysis is done at a collective level, not aiming at putting individuals in certain categories. An individual can hold several of the understandings expressed in the categories of description, but mapping between individuals and categories is not the aim of the analysis. It is unlikely

that the collected data can reveal all the different ways in which each individual student understands the concepts of interest. However, when statements from different students are brought together, that collective "pool of meaning" reveals a rich variety in understandings. When quotes are taken out of their contexts and compared to each other, the individuals are put in the background, and the collective understandings of the group are in the foreground.

Learning is understood as developing richer ways to see a phenomenon, as represented in the more advanced categories of the phenomenographic outcome space. Variation theory, which originates from phenomenography, emphasises *variation* and *discernment* as key words in this process. A necessary but not sufficient condition for discerning a specific feature of a phenomenon is that the student gets the opportunity to experience variation in a *dimension* corresponding to that feature. In Paper IV we explain *dimension of variation*, or for short dimension, in the following way:

> For example, if 'size' and 'colour' are the features of a phenomenon 'picture component', then there is a 'size' dimension and a 'colour' dimension of the corresponding feature space. A particular instance of 'picture component' can be represented by its values in those dimensions, i.e., by its particular size and colour.

Each feature of the phenomenon studied that appears in an outcome space corresponds in this way to a dimension. Marton, Runesson and Tsui (2004, p. 21) discuss the need to create a space, which means "opening up a dimension of variation (as compared to the taken-for-granted nature of the absence of variation)." The authors describe such a space:

> A *space of learning* comprises any number of dimensions of variation and denotes the aspects of a situation, or the phenomena embedded in that situation, that can be discerned due to the variation present in the situation. [...] [The space] delimits what can be possible learned (in sense of discerning) in that particular situation. (Marton et al., 2004, p. 21) (Italics in original)

3.3 Content analysis

Content analysis is described by Mostyn (1985) as "a very ordinary, everyday activity we all engage in [...] when we draw conclusions from unstructured communications" (p. 115) Content analysis originally dealt with quantitative analysis of data (*what*-, *where*-, and *how many*- questions) but has developed to include qualitative analysis (*why*-questions).

Qualitative content analysis as a research method deals with analysing artifacts, often texts, with focus on the content and meaning embedded in the text. The goal of qualitative content analysis is to understand the meaning of unstructured communication, and through a process of condensing raw data into categories come to a better understanding of the phenomenon studied. This

process involves inferences and interpretations that require knowledge of the context and subject studied. Hsieh and Shannon (2005) define qualitative content analysis in the following way:

> a research method for the subjective interpretation of the content of text data through the systematic classification process of coding and identifying themes or patterns (Hsieh and Shannon, 2005, p. 1278)

Mostyn describes qualitative content analysis as "the 'diagnostic tool' of qualitative researchers" (p. 117). As such, content analysis is used in a variety of research methods including discourse analysis, ethnographic research, and computer text analysis (Krippendorff, 2004, p. 19).

The object of interest in qualitative content analysis is often a text, for example transcribed interviews, but data can equally come from observing behavior, artifacts, etc. (Mostyn, 1985, p. 124). To gain insights into the meaning embedded in the data, the analysis requires interpretation that goes beyond inference. Mostyn writes:

> we become concerned with content as a reflection of deeper phenomena. Words are treated as symbols and the data has attributes of its own; we are analyzing both manifest and latent data. (p. 116)

Graneheim and Lundman (2004) describe manifest as "what the text says [...] the visible, obvious components", while latent data is described as "what the text talks about [...] an interpretation of the underlying aspects of the text". (p. 106)

The researcher scrutinizes the data, looking for regularities "in terms of single words, themes, or concepts." (Mostyn, 1985, p. 118) This in-depth analysis leads to identification of *categories*, which is the heart of content analysis.

Mayring (2000) describes content analysis as "a bundle of techniques for systematic text analysis". He specifically describes two approaches that offer procedures for data analysis. The first is *inductive category development*. It is described as a "reductive" process:

> the material is worked through and categories are tentative and step by step deduced. Within a feedback loop those categories are revised, eventually reduced to main categories [...]

In this way, data can be categorised with an explorative approach. The categories are developed as the researcher delves deeper into the data and lets the data speak. The categories developed during the process guide the researcher in his or her interpretations and inferences in the analysis.

The second approach is *deductive category application*. Mayring writes: "Deductive category application works with prior formulated, theoretical derived aspects of analysis, bringing them in connection with the text."

Meaning embedded in the data is, with this approach, unveiled when parts of the data are fitted into pre-existing categories or theories, and the result

subsequently is interpreted. The researcher uses his or her knowledge of the data in terms of the participants and context of the data gathering in the process of interpretations and inferences.

3.4 Trustworthiness in Qualitative Research

Evaluation of research and its trustworthiness depends on the research paradigm used. This is due to the fact that different research paradigms have different knowledge claims. Lincoln and Guba (1985) write that "criteria for what counts as significant knowledge vary from paradigm to paradigm." (Lincoln and Guba, 1985, p. 301)

Lincoln and Guba contrast criteria related to what they call the conventional paradigm, where most research in the area of computer science belongs, with criteria appropriate for the naturalistic paradigm, where the present, qualitative research belongs. The trustworthiness criteria of the conventional paradigm are often discussed in terms of "internal validity", "external validity", "reliability", and "objectivity". Internal validity "refers to the extent to which the findings accurately describe reality" (Hoepfl, 1997, p. 58), and external validity "refers to the ability to generalize findings across different settings." (Hoepfl, 1997, p. 59). Hoepfl writes with reference to Kirk and Miller (1986, p. 41-42), that three different types of reliability have been identified in conventional research:

> 1) the degree to which a measurement, given repeatedly, remains the same; 2) the stability of a measurement over time; and 3) the similarity of measurements within a given time period (Hoepfl, 1997, p. 59–60)

Lincoln and Guba write that the usual criterion for objectivity is "intersubjective agreement; if multiple observers can agree on a phenomenon their collective judgment can be said to be objective." (Lincoln and Guba, 1985, p. 292) Another approach to establish objectivity is "through methodology; to use methods that by their character render the study beyond contamination by human foibles." (p. 292–293)

The comparable criteria in the naturalistic paradigm are "credibility", "transferability", "dependability", and "confirmability" (Lincoln and Guba, 1985, p. 300).

There have been several phases in the development of qualitative research, and the discussions in the literature on how to certify trustworthiness have consequently developed over the years.

In the present discussion on evaluation of qualitative research I will use the trustworthiness criteria suggested by Lincoln and Guba (1985). There are other approaches suggested to ensure trustworthiness in qualitative research, for example applying ideas from the conventional paradigm criteria. A recent example of a discussion on trustworthiness as an alternative construct to validity, reliability, and generalisability in phenomenographic research is Collier et al. (2008).

In the following I will discuss each of the four criteria; credibility, transferability, dependability, and confirmability as they, according to Lincoln and Guba, can be used to evaluate trustworthiness of research within the naturalistic paradigm.

The first criterion, *credibility*, deals with carrying out an inquiry in a way that enhances the chances for the findings to be found credible or believable. This can involve credibility from the participants' perspective. Credibility has less to do with the size of the sample than the quality of the data, the analysis and the written report. Lincoln and Guba suggest techniques to address credibility. Examples of these techniques are prolonged engagement, persistent observations, peer debriefing and triangulation, where the latter can refer to triangulation of data, methods, multiple analysts, and theory.

The second criterion, *transferability*, refers to the degree to which the results of the research can be transferred to other settings, contexts, or populations. Lincoln and Guba discuss transferability in terms of where "the burden of the proof lies" (Lincoln and Guba, 1985, p. 298). Instead of making generalizations, the researcher should provide what Lincoln and Guba call a "thick" description of the research. This description can include description of preparation of the study and the underlying research questions and assumptions, data gathering including choice of data collection methods and participants, quotations from interviews, analysis methods and decisions taken during the process of analysis, and the inferences the researcher has come to, and more. The thick description is presented to the reader so that he or she can determine whether the findings are applicable to his or her situation. The investigator does not know the context of the receiver. Only the receiver of the research knows and can judge the transferability of the research. Lincoln and Guba write:

> The best advice to give to anyone seeking to make a transfer is to accumulate *empirical* evidence about contextual similarity; the responsibility of the original investigator ends in providing sufficient descriptive data to make such similarity judgments possible. (Lincoln and Guba, 1985, p. 298) (Italics in original)

The authors emphasise that the researcher should provide "the thick description necessary to enable someone interested in making a transfer to reach a conclusion about whether transfer can be contemplated as a possibility." (Lincoln and Guba, 1985, p. 316)

The third criterion discussed is *dependability*, which is the naturalistic correspondence to reliability. Dependability emphasises that the researcher needs to account for the changing context in which the research occurs. Lincoln and Guba write: "The naturalist sees reliability as part of a larger set of factors that are associated with observed changes." Aiming to demonstrate dependability "the naturalist seeks means for taking into account both factors of instability *and* factors of phenomenal or design induced change." (p. 299) Lincoln and Guba write that it is argued that if credibility is fulfilled, dependability is also fulfilled: "Since there can be no validity without reliability (and thus no credibility without dependability), a demonstration of the former is sufficient

to establish the latter." (p. 317) Techniques related to credibility thus ensures that dependability is fulfilled.

The fourth criterion is *confirmability*, which corresponds to objectivity in the conventional paradigm. To what degree can the results of the research be confirmed or corroborated by others? Lincoln and Guba write that there are three different perspectives on objectivity (p. 299). The perspective that is often preferred by naturalists is "Objectivity exists when an appropriate methodology is employed that maintains an adequate distance between observer and observed." (p. 300) The authors conclude that this definition

> removes the emphasis from the investigator (it is no longer his or her objectivity that is at stake) and places it where, as it seems to the naturalist, it ought more logically to be: on the data themselves. [...] Are they or are they not *confirmable?* (p. 300) (Italics in original)

Techniques to ensure confirmability suggested by Lincoln and Guba are for example triangulation and the keeping of a reflexive journal. (p. 319)

4. The present research

This section discusses how the research approaches presented in Section 3.2 and Section 3.3 are applied in the present thesis. Subsequently the section discusses how trustworthiness, as discussed in Section 3.4, has been ensured in the different investigations involved in the thesis.

4.1 Research approaches applied in the present research

The thesis has three themes as is discussed in Section 1.1. The first theme, student learning of concepts, is analysed by means of deductive content analysis (Paper III), by means of phenomenography (Paper I, Paper IV, and Paper V), and by means of variation theory (Paper I and Paper IV). Paper II is mainly a literature review of work related to "threshold concept" (Meyer and Land, 2005) and will not be discussed below.

The second theme, student learning of practise, is analysed by means of inductive content analysis (Paper VI and Paper VII).

The third theme, the relationship between students' conceptual and practical learning, is analysed by means of deductive content analysis (Paper VIII), and by means of phenomenography and variation theory (Paper IX).

4.1.1 Phenomenography and variation theory in the present research

Paper I includes a traditional phenomenographic analysis of novice students' understanding of two central concepts in object-oriented programming, object and class. The analysis is described, and quotes from the students illustrate the different understandings identified. Two sets of categories of description are presented, and features of the different understandings are identified. The paper further includes a discussion of how variation theory can be applied to the phenomenographic results, and implications for teaching are inferred from the latter discussion.

Paper IV includes a traditional phenomenographic analysis of students' understanding of the phenomenon "programming" including quotes that illustrates the different categories of description. The paper further shows how the phenomenographic results can be used to design learning activities that support students' learning, by use of variation theory and patterns of variation. To this end we introduce the theory of phenomenography and variation theory,

and give a thorough review of the steps taken to identify the dimensions of variation that are related to the phenomenographic outcome space discussed. Finally we suggest appropriate patterns of variation that can be used to help students discern some of the identified dimensions of variation, and how these patterns can be applied in teaching novice programming.

Paper V includes a traditional phenomenographic analysis of students' understanding of the phenomenon "learning to program" with a description of the analysis performed and with quotes from the students to illustrate the categories of description identified in the analysis. We relate the results from the analysis to the "process-object duality" theory from mathematics education. We show that the phenomenographic analysis reveals problems students experience in learning object-oriented programming, not indicated in the "process-object duality" theory.

Paper IX builds mostly on the results presented in Paper I, Paper IV, Paper V, and Paper VIII. Paper IX uses phenomenography and variation theory to build an analytical model of students' learning of practise and concepts. Dimensions of variation are in the center of the discussion, tying together students' conceptual and practical learning. The conceptual understandings used to illuminate the analysis are novice students' different understandings of the concepts object and class and related dimensions of variation, as presented in Paper I. The practise is analysed by identifying common novice programming activities at different level of proficiency. Subsequently it is argued that these activities also are related to the same dimensions of variation. In this way practise, expressed as activities at different level of proficiency, and qualitatively different conceptual understandings are related through dimensions of variation, and a model of the complex learning process of novice programming is developed.

4.1.2 Content analysis in the present research

Paper III is based on deductive content analysis of semi-structured interviews with students from the second investigation who discuss important and difficult concepts they have met during their education. The interview questions were constructed to capture Meyer and Land's definition of threshold concepts (Meyer and Land, 2005). For each concept discussed by the students we analysed whether the criteria that characterize threshold concepts were met. In this way we identified two threshold concepts in computer science, pointers and object-oriented programming.

In Paper VI inductive content analysis is used on parts of the interviews discussed above. The goal with the research was to identify strategies that students use successfully in their computing studies, and to categorize the strategies in ways that made them useful for future students and educators (Paper VI, p. 156). Some interview questions concerned what the students did when they were stuck in their learning process. We aimed at identifying all strategies to get unstuck mentioned by the students. In an iterative process we grouped the strategies in categories. We first created many small cate-

gories where only a few strategies that appeared more or less the same were grouped together. The smaller categories were subsequently grouped together into broader, more abstract categories that covered large numbers of related strategies. The categories, the many small as well as the broader and more abstract, are presented in the paper.

Deductive content analysis is used in paper VIII on the interview data from the second investigation. Meyer and Land (2005) use the metaphor *the liminal space* to capture important features of students' experiences of being in the midst of learning threshold concepts (Meyer and Land, 2005). We analysed the parts of the interviews where students discussed their learning of the threshold concepts identified in Paper III. The analysis aimed at investigating the liminal space criteria, as discussed by Meyer and Land, and discussing how these criteria appear in computer science students' learning process.

In Paper VII, inductive content analysis was used to categorise artifacts produced by senior students. The research examined how students' software designs can be compared, and in addition investigated senior students' ability to design. The students were asked to design a software system. The designs, made on papers, were the artifacts analysed by the research group. Starting with a sample of 20 of the total 149 designs, we made an initial categorisation of the designs, based on their semantics and guided by our experiences as computer scientists, researchers, and teachers. We subsequently categorised the remaining designs, each researcher categorising 70 designs. Agreement was reached in close discussions within the research group. The identified categories were subsequently discussed in relation to academic and demographic background data gathered from the participants, which supported the interpretation of the results of the categorisation. In this part of the analysis we used quantitative methods for parts of the interpretation of the categories and inferences drawn.

4.2 Trustworthiness of the present research

In this section I will discuss how trustworthiness, as presented in Section 3.4, have been ensured in the present thesis. The three investigations will be discussed separately. For each of the three criteria suggested by Lincoln and Guba (1985), credibility, transferability, and confirmability, I will discuss how I have used techniques to ensure trustworthiness in the investigations. Since the dependability criterion is fulfilled with the credibility criterion, dependability will not be discussed (Lincoln and Guba, 1985, p 317). The techniques I discuss relate to the techniques suggested by Lincoln and Guba, but with minor modifications in my applications of them.

4.2.1 Trustworthiness in the first investigation

Data in the first investigation were collected from a series of interviews with 14 first year students who had just finished their first programming course.

The interviews aimed to elicit students' different understandings of some central concept. Research approaches used on the data are phenomenography in Paper I, Paper IV, Paper V, and Paper IX, and variation theory in Paper I, Paper IV and Paper IX. Below I describe how I have established trustworthiness in the phenomenographic analyses which includes variation theory, by fulfilling the three criteria.

Data collection and the analysis of the data from the first investigation was done in close discussion with a colleague. Both of us have long experience of teaching programming, and can thus be said to have prolonged engagement in the field. The phenomenographic analysis was partly performed by me, and later scrutinised by my colleague, and partly done by both of us separately, and then joined in discussions where we came to agreement on the results. The analyses have further been discussed in seminars with researchers in the CER field, and conference and journal papers have been peer reviewed. Peer debriefing has thus been used and the credibility criterion ensured.

The second criterion, transferability, has been ensured in the presentations through a thick description in the following ways. Great effort was made to ensure that the group of students investigated, the choice of participants for the interviews, the course they studied, the questions asked, the research approach taken, and the analyses performed were described in as much detail as possible considering page limitations and other practical limitations. In this way a thick description was provided for the reader.

Keeping of a reflexive journal is one technique suggested by Lincoln and Guba to ensure the confirmability criterion (p. 319). In the first investigation reflexivity, as discussed by Finlay (2002), has been used to some extent to ensure confirmability. The researcher always influences both collection and interpretation of data. Reflexivity means explicitly, with self-awareness, analysing one's own role in the research process (Finlay, 2002). Finlay writes: "Reflexive analysis in research encompasses continual evaluation of subjective responses, intersubjective dynamics, and the research process itself." (p. 532) This has been carried out in close dialog with my co-authors, by presenting the research at conferences, and specifically in educational situations where people from outside the phenomenographic community have been introduced to this research approach partly through my own research. This has encouraged me to analyse my own role in the research process. In this way the third criterion, confirmability, has been ensured.

4.2.2 Trustworthiness in the second investigation

The second investigation was multinational. This implies that the pool of data is possibly richer than if the data came from one institution only. The background of the participants is more diverse, and the education differs between countries and institutions. To ensure stringent interview conditions so that the data can be treated as one pool of information, certain steps were taken concerning choice of participants, and preparation of the interviews. To ensure that the participants were at comparable level in their educations, and stud-

ied at similar study programs, the issues of the background of participants and their study programs were thoroughly discussed within the group of researchers. To ensure trustworthiness in the performance of the study, the interview questions were worked out in close discussion among the researchers. One of the researchers subsequently performed a pilot interview with the rest of the researchers present, listening, but not interfering. The pilot interview was followed by a discussion among the researchers to resolve possible questions. The pilot interview was not added to the pool of information used in the subsequent analyses. The purpose of the pilot interview was solely to ensure trustworthiness in the performance of the study.

Research approaches used on the data are deductive content analysis in Paper III and Paper VIII, and inductive content analysis in Paper VI. Below I describe how my use of content analysis ensured trustworthiness in ways that fulfill the criteria discussed.

All researchers involved in the second investigation have long experience of teaching computer science. In addition, the work has been triangulated concerning data gathering methods in the following way: we have performed informal interviews with educators, an instructor survey, interviews with students, and a literature survey. Prolonged engagement in the field and triangulation ensures the fulfillment of the first criterion, credibility. Triangulation furthermore ensures the third criterion, confirmability.

In our reports we provided thick descriptions of interviewees, interview and survey questions, excerpts from the data, and how the analyses were performed. This ensures transferability.

4.2.3 Trustworthiness in the third investigation

The third investigation was performed by a group of 21 researchers from four countries. The investigation was led and designed by three of the researchers. Data was gathered by all researchers. A multinational study implies rich data in the sense described above. To ensure stringent interview conditions so that the data can be treated as one pool of information, certain steps were taken concerning choice of participants, and preparation of the "design brief", that is the task given to the participants which describe the problem and provide instructions (see Paper VII, p. 198). All researchers were given the same, detailed information on what was regarded as appropriate level of education of expected participants. The performance of the study was described in detail, and before the study was performed all researchers tried the "design brief" themselves, followed by a discussion among the researchers. In this way the researchers were given an opportunity to discover ambiguities in the setup of the study, and resolve differing understandings among themselves. The "design brief" was thereafter reviewed by the leaders of the investigation in line with the responses from the researchers before given to the participants.

The research approach used in Paper VII is deductive content analysis, which, as described below, has been used in a way that ensures trustworthiness.

As in the second investigation, prolonged engagement in the field is fulfilled since all researchers involved in the data gathering as well as those involved in the analysis of the data have long experience of teaching computer science. The subset of data used in Paper VII, designs from senior students, is triangulated consider that the participants came from 21 institutions in four countries. In this way the first criterion, credibility, as well as the third criterion, confirmability, have been fulfilled.

The report of the research provides the reader with a thick description which ensures transferability. The study is clearly described, the "design brief" given to the students is included in the paper, and the paper gives a rich description of the categorization procedure performed.

5. Results

This section presents results from the nine papers included in the thesis, organised around its three themes. The first theme, *student learning of concepts*, form a major part of the thesis. There is a large body of previous research on the second theme, *students learning of practise*, and the present research on this theme thus focuses on one specific skill, software design, which is less well researched than other central skills like reading and writing code. In the third theme, *the relationship between conceptual and practical learning*, I synthesise the results from the first two themes using the conceptual framework Ways of Thinking and Practising.

Results concerning possible implications for teaching are brought together in Section 6.

5.1 Learning of concepts

The first theme deals with problems concerning student learning of concepts. In particular three specific aspects of student learning of concepts are researched, reflecting the research questions in Section 1.1.

The first research question presented in Section 1.1 concerns novice students' understanding of programming concepts. The phenomenographic analysis presented in Paper I shows that the novice students' understanding of the concepts object and class vary from a narrow textual representation of the concepts, to a broader understanding of the concepts including the active behaviour of the objects when the program is executed, to the most desirable understanding that includes the two previous, but also the modeling aspects of the concepts. Very few students seem to have reached this understanding although it is fundamental in object-oriented programming. In particular our results show that:

- There are particular ways to understand the concepts *object* and *class* that are critical for students to discern. Few students seem to have reached the full understanding of the concepts described in the phenomenographic outcome spaces. The results from Paper I can be used by educators in the way that they can accentuate the identified features of the concepts in their teaching, and thus facilitate for students' learning and further studies.

The investigation presented in Paper I of students' understandings of the concepts object and class is in line with results presented in Paper II and Paper III, where conceptual learning is discussed at a higher level of granularity. While the first paper focuses on two specific concepts in the object-oriented

paradigm, namely object and class, Paper II and Paper III aim at discussing and identifying important concepts, so called "threshold concepts" (Meyer and Land, 2005) in computer science in general. Threshold concepts are described by Meyer and Land as a subset of core concepts in a discipline that might be used to organize and focus education.

Paper II discusses threshold concepts in relation to a number of other computer science education research areas. Paper II specifically discusses how the idea of threshold concepts relates to, and differs from, constructivism, mental models, student misconceptions, breadth-first approaches to introductory computer science, and fundamental ideas. We found support in the literature for the conclusion that abstraction and object-orientation fulfill the criteria of being threshold concepts.

Paper III continues the discussion started in Paper II. The paper presents an empirical investigation that aimed at identifying possible threshold concepts in computer science. We found empirical evidence of two threshold concepts: object-oriented programming, which was suggested from the literature survey in Paper II, and pointers.

We have not yet pinpointed which aspect(s) of object-oriented programming is(are) threshold concept(s). Object and class, which are the concepts investigated in Paper I, seem however to be reasonable candidates to study since they are not only central, but often introduced early in programming education. In particular our results show that:

- There exist threshold concepts in computer programming. The identification of such concepts gives valuable information for educators. Object and class, which are threshold concepts candidates, can act as nodes around which introductory programming education can be organised.

Paper IV and Paper V investigate student understanding of what programming and what learning to program means. This is the second research question.

The action dimension described in the outcome spaces in Paper I reappear in the outcome space in Paper IV. There seems to be similarities between how the phenomena "object" and "class" are understood relative to the phenomenon "computer programming". Or to phrase it with the variation theory terminology: there are dimensions in the programming learning space that seem to be common to several phenomena. If a specific feature of one phenomenon is discerned, for example a feature related to the action dimension, this can facilitate for the learning of other phenomena which are related to the same dimension. This is further developed in Paper IX, where variation theory is used to research students' learning process.

Another interesting connection between Paper I, Paper IV, and Paper V is the similarities in their respective outcome spaces. The understandings described go from a narrow, programs-as-text and programming-as-coding foci, to broader understandings that include the reality outside the computer and the course. The importance of students reaching the more advanced ways of understanding these phenomena are pointed to for example in Computing Curricula 2001 Section 7.2 (Roberts and Engel, 2001).

Students' understandings of the subject studied, here computer programming (Paper IV), *and* their understanding of what it means to learn that subject (Paper V), have been shown to be important for how students approach their studies, and thus have impact on the learning outcome (Booth, 1992, p. 261–262). Paper IV consequently discusses how variation theory, and specifically the patterns of variation discussed by Marton and Tsui (2004), can be applied to a phenomenographic outcome space. The discussion gives examples on how a phenomenographic analysis can be applied in teaching to help students advance their understanding of what computer programming means, which is the third research question.

In particular our results show that:

- There is a wide variety in students' understanding of what programming and learning to program means. Related work has emphasised the importance of students coming to a good understanding of what programming, and learning to program means. The phenomenographic outcome spaces and variation theory can be used by educators to facilitate for students' learning of these matters.

Paper V points to similarities with Hazzan's (2003) work on the process-object duality learning theory. Paper V however also points to differences in terms of problems novice students encounter, which are not observed in the process-object duality theory. The practise, discussed as processes by Hazzan, might be the major obstacle for some programming students in their learning. If the practise is experienced as too difficult to master, it can not serve as the a means for students to reach the learning goals. In this way results from Paper V point to the next theme of the thesis, students' learning of the practise.

5.2 Learning of practise

The second theme concerns the role of practise in computer science education. Practise is important for students' learning to program, both as a means to reach the learning goals, and as a learning goal in itself. Paper VI focuses on the former aspect in terms of students' learning strategies, which is the first research question in the second theme of the thesis, see Section 1.1. Paper VII discusses the latter aspect of practise, namely one specific learning goal, software design. This reflects the second research question in the second theme of the thesis. Software design is one of several important skills programming students are supposed to learn, as discussed in Paper IX.

Students' approaches to their learning have shown to be important for the learning outcome (Marton et al., 1984). According to for example Trigwell et al. (1994), approaches to learning can be discussed in terms of intention and strategy, where intention is what the student attempts to do, while strategy is what the student does to fulfill the intention. Paper VI identifies students' learning strategies. We discuss strategies in terms of what the students said they did when they were stuck in their learning.

The list of identified strategies is long, altogether 35 strategies. We grouped the strategies into four super-categories. The super-categories found are In-

puts/interactions where the students talk about getting help from elsewhere, Concrete/do stuff which is in line with what I call learning through practising, Abstract/understand stuff which captures when the students discuss "learning and getting unstuck at a higher level" (p. 158), as opposed to the former category, and the last category, "Use the Force", which involves strategies where students use "their willpower or character" (p. 158).

Programming strategies have been claimed to be important parts of programming skills (Davies, 1993). Robins et al. (2003) found in their literature review that lack of programming strategies caused problems for novice students. Our analysis is focused on learning strategies. Davies' and Robins et al.'s discussions are still relevant for our research since the strategies discussed in our study often concerns problems with programming concepts.

Although all students in the study but one described that they sometimes were stuck in their learning, they all had several strategies for getting unstuck, and the strategies were surprisingly diverse.

In particular our results from the research on practise as a means to reach learning goals, show that:

- It is important that students learn a variety of strategies for mastering the problems they meet in their learning. We specifically noticed the "importance of social interaction, and the active responsibility taken by the students". (Paper VI, p. 160)

Paper VII discusses one particular programming practise, software design. The focus of the paper is the problem of assessing designs: how can students' software designs be analysed and compared? The focus of the investigation and the data collection, and my focus in this theme, is however on how students learn to design. This is also present in the paper, but given as a background for the discussion on how to analyse rich artifacts like written and drawn designs.

The paper shows how the technique of semantic categorization can be used to organize such rich artifacts. 149 designs produced by near-graduating students were categorized into six groups of similar designs, depending on their semantics, and "ordered relative to the degree to which the stated requirements were met" (p. 199).

The overall question of the study was: Can students near graduation design software systems? Only 9% of the designs were assessed as reasonable designs. They fell into the two highest categories, labeled Partial design and Complete, of which only 2% were Complete. 29% of the designs fell into the category First step and showed some progress toward a design. The remaining 62% had no or very little information added beyond the specification given.

We saw a significant increase in number of syntactic features in the higher categories compared to the lower. Also the length of the designs increased in the higher categories, except for the highest, which slightly decreased, and the more advanced designs more often than the others included "an overview, details on part responsibilities, and communication between the parts" (p. 199). Other observations of interest are that there was a positive correlation between number of computer science courses taken and the category of the design: the more courses taken the higher category; academic performance measured by

grades on computer science courses seemed though to have little or no relationship to the design produced.

We found that semantic categorization is possible, but time-consuming and some designs required extensive discussions between the researchers to be categorized.

In particular our results from the research on practise as a learning goal show that:

- Computer science students near graduation have great difficulties in mastering design tasks, even though design is one of the core skills the students are supposed to learn during their education.

The present research points to the problematic role of practise in programming students' learning. Practise is not merely the means to reach the conceptual learning goals. The role of practise in the learning process needs to be further researched.

5.3 Ways of Thinking and Practising

The research presented in Paper VIII, From *Limen* to *Lumen*: Computing students in liminal spaces, investigates the learning process related to threshold concepts in computer science, which is the first research question in the third theme in the thesis. Paper III identifies two such concepts, pointers and object-oriented programming. In Paper VIII we take the analyses from Paper III one step further. We use the theory of *liminal space* as it is discussed by Meyer and Land (2005). The liminal space is described as "the transitional period between beginning to learn a concept and fully mastering it" (Paper VIII, p. 124). We applied the theory to our data as a framework to highlight certain features of the learning experience which, looking at the data as a whole, are difficult to discern. We looked for the standard features of the liminal space as described by Meyer and Land, but in addition we found some that may be specific to computer science. The result of the analysis reveals a broad and rich picture of the students' learning experiences.

The picture thus unfolded shows a transformative process which often takes long time, involves strong emotions and elements of mimicry, and contains specific parts, or sorts of understandings, which can become stuck places for students. The parts identified include an abstract, or theoretical, understanding of the concept; a concrete understanding – the ability to implement the concept (without necessarily having the abstract understanding); the ability to go back and forth between the abstract and the concrete understandings; an understanding of the rationale for learning and using the concept; and an understanding of how to apply the concept to new problems. Students need to attain all these understandings. This can explain why they obviously get stuck at different places, and "why the path through this space is not a simple linear progression." (p. 130) Students rather seem to "need to go back and forth between the theoretical and the practical" (p. 130), and different students take different "routes" depending on individual stuck places. We further point

to a result characteristic for computer science: "we commonly observed the particular partial understanding of not being able to translate from an abstract understanding to concrete implementation or design" (p. 130)

Beside this we discuss students' expressions of how, and if, they know that they know a concept. The students express the experience of mastering a concepts sometimes as emotional, sometimes as being able to visualize their understanding, and sometimes as being able to master the handicraft of programming. We further found evidence in the data that there were students who said they knew a concept that they apparently did not fully know, and students who doubted their own knowledge even though it seemed as if they knew.

The learning is experienced as a complex whole by the students, and thus difficult to fully discern. We found that the liminal space, as an analytic tool, provided a way to theoretically separate several important features of the process, and thus untangle some of the complexity of the learning.

In particular our results give empirical evidence that:

- The practise as well as the concepts are problematic for the students to learn. Both are important in the learning process and can become stuck places for the students. If the students face a problem with one of them, it is expected to have negative influence on the other.

In this way Paper VIII gives an empirical background for the analysis presented in Paper IX.

Paper IX, Ways of Thinking and Practising in Introductory Programming, is the synthesis of my thesis work. The paper builds on results from the first and the second investigations. Students' conceptual and practical learning are investigated, specifically how practise and concepts relate in student learning, which is the second research question in the third theme of the thesis. I argue from the empirical data that concepts and practise are equally important parts of the learning goals, and equally difficult for students to learn. Furthermore, there is a mutual dependency and complex relationship between the two. This discussion points to the need to research this relation. In particular:

- The research identifies dimensions of variation related to qualitatively different conceptual understandings. The research further identifies dimensions of variation related to practise in terms of programming activities at different levels of proficiency.
- Based on results from the analyses of the data and elements from phenomenography and variation theory, an analytical model is proposed. The model shows that activities as well as conceptual understandings relate to dimensions of variation.

Previous research has discussed dimensions of variation related to concepts as well as to practise (Marton and Tsui, 2004; Fazey and Marton, 2002). The present research takes this one step further. In particular:

- The most significant finding is that practise, in terms of programming activities, and conceptual understandings have dimensions of variation *in common*. This was possible to show since the research proposes a way to identify dimensions of variation related to practises.

- The dimensions of variation are thus like interfaces between conceptual understandings and activities. If a dimension of variation is discerned, this can open a possibility for students to discern concepts *and* to learn activities in new ways.

This finding can to some extent explain the complex learning of computer programming, where some students seem to first learn the concepts and then the practise, while other students seem to learn in the opposite order. The model can further to some extent explain why programming activities not always facilitate for students' learning. If the activity is at a level of proficiency that presupposes dimensions of variation not yet discerned by the student, the student might have problems to learn through the activity. This can be phrased using terminology from variation theory: if the patterns of variation involved in the learning situation are too complex, students might not discern the dimensions of variation involved in the situation.

The result shows that the dimensions of variation can relate to several concepts and activities. This carries implications for learning, in particular:

- If, for example, one way to understand a concept is discerned through a dimension of variation, this learning experience can facilitate for discernment of other related ways to understand concepts, and for the learning of related activities.

The results also indicate that activities as well as concepts can be related to more than one dimension of variation. It is fundamental in variation theory that concepts can relate to more than one dimension of variation (Marton and Tsui, 2004). The result that activities can relate to more than one dimension of variation indicates in particular that:

- Higher level of practical proficiency relate to more dimensions of variation in a similar way as more advanced ways to understanding concepts relate to more dimensions of variation.

6. Discussion - from a teaching perspective

As educators we know that many students, specially the novices, have great difficulty learning to program. This section will discuss how results from analyses inspired by phenomenography and variation theory can be implemented in programming teaching to facilitate for students' learning.

I will first discuss and develop the phenomenographic analysis presented in Paper I, see Section 6.1. This discussion is inspired by the research presented in Paper IV. In Section 6.2 I further discuss implications for teaching emanating from the analysis on how students' conceptual and practical learning relate, which is presented in Paper IX.

6.1 Phenomenography in practise - an empirical example

To exemplify how a phenomenographic outcome space can be used by educators, I will show an outcome space of novice students' understandings of the concepts object and class, and discuss a process that starts with a phenomenographic outcome space, identifies critical features of the phenomena (in this example two object-oriented concepts), discusses corresponding dimensions of variation, and arrives at implications for teaching in terms of concrete advice for educators. For a comprehensive description of the data and the analysis that gave the outcome space, see Paper I and Eckerdal (2006).

The results presented in Paper I indicate that many students have a problem fully grasping the investigated concepts object and class. The phenomenographic outcome spaces give however valuable information to educators on the different ways in which our students can understand these concepts. In addition the outcome spaces provide information necessary for retrieving what is *educationally critical* for a good understanding of the concepts. Educationally critical means that there are certain ways to understand a concept that are critical in the sense that if the student has not discerned these ways of seeing the concept, something important is missing, something that might be critical for the students future studies, or critical for the development of the student's capabilities in the subject studied, here computer programming. Educationally critical features of a concept can be identified by use of other techniques, see for example Runesson (2006), but in the present thesis phenomenography has been used.

6.1.1 The phenomenographic outcome space

The concepts object and class are closely related, and can hardly be understood without each other. When describing the different understandings found in the data, it is not surprising to find similar patterns for the understandings of the two concepts. The initial two outcome spaces presented in Paper I have thus been collapsed in one outcome space in Table 6.1 below.

Class is understood as an entity of the program, contributing to the structure of the code and describing the object, where the object is understood as a piece of program text.
As above, and in addition class is understood as a description of properties and behaviour of objects, where object is understood as something that is active during execution of the program.
As above, and in addition class is understood as a description of properties and behaviour of objects, where object is understood as a model of some real world phenomenon.

Table 6.1: *Summary of categories describing the different ways to understand the concepts* object *and* class *found in a group of novice students. The latter categories include the understandings in the former.*

The analysis indicated inclusive categories, as expressed in Table 6.1. This means that an understanding expressed in one of the latter categories includes the understandings expressed in the former, and thus expresses a richer understanding of the concepts. It is hardly possible to understand that an object is a model of something in reality without understanding that this implies a description of its properties and behaviors, expressed in the code.

What can we as educators do to facilitate for the students to develop their conceptual understanding? The following three sections, inspired by the research presented in Paper IV, discuss how the empirical results presented in Table 6.1 can be further analysed and give implications for teaching.

6.1.2 Discernment and variation - identification of critical features

As discussed in Section 3.2, different categories in an outcome space represent combinations of features of the phenomenon, which are present in the focal awareness at a particular point in time (Marton and Booth, 1997, p. 126).

Learning is understood as developing richer ways to see the phenomenon, as represented in the more advanced categories of the phenomenographic outcome space. A necessary, but not always sufficient condition for discerning a specific feature of a phenomenon, is that the student gets the opportunity to experience variation in a dimension corresponding to that feature. (Marton et al., 2004, p. 31).

The first category in Table 6.1 reflects the students' understanding of classes as entities of the program, contributing to the structure of the code, and objects as a piece of program text. The focus of this understanding of a class is the appearance of the structure of the program text. The focus of the understanding of objects, is on the program text. The feature, critical in this category is thus the textual representation of the concepts.

In the second category, in addition to the above understanding, classes are understood as descriptions of properties and behaviour of objects, where objects are understood as something active in the program. The focus in this category is on what happens during execution of the program, in particular on the objects created and how they contribute to different events at run-time[1]. The objects are the active parts of the program, accomplishing the task given. The new feature added to this category is the active behavior when the program is executed.

The last category includes, in addition to what is described above, that classes are understood as descriptions of properties and behaviour of objects, where objects are understood as models of some real world phenomenon. The focus is still on the class' description of the active objects, but now with an emphasis on the reality aspect of the class description. The new feature expressed in this category is the modeling aspects of the concepts.

The students' foci, and consequently the critical features of the concepts, are hence identified. Variation in a dimension corresponding to a feature is, as discussed above, a prerequisite for learning to take place. Having expressed the identified critical features of the concepts, as captured by the categories of description in Table 6.1, it is now possible to discuss what dimensions of variation correspond to each feature.

6.1.3 Dimensions of variation - open a space for learning

When there is a variation in a dimension that corresponds to a critical feature, this opens a possibility for students to discern the feature and thus learn the concept in a new way. In the first category in Table 6.1 the critical feature is the textual representation of the concepts. To be able to discern this feature, students need to discern that in different programs objects and classes appear in different ways. In that sense, the textual representation of programs constitutes a relevant dimension related to this feature. Different, specific program

[1]For readers not familiar with programming: "run-time" means the period of time when a program is running.

texts constitute values along this dimension and if students discern such variation, it opens the possibility of understanding object and class in this way.

The new feature expressed in the second category that the students need to discern is the active behavior of the program during execution. Different actions resulting from different program executions constitute values in the corresponding dimension of variation.

In the last category in Table 6.1, the new feature added is the modeling aspect of the concepts. In this case, different real-life phenomena modeled as classes and corresponding objects, constitute values along this dimension.

The line of reasoning above is summarized in Table 6.2. It includes the students' different understanding of the concepts object and class, as expressed in Table 6.1, see the left column in Table 6.2. The right column includes the corresponding dimensions of variation.

Students' understandings of the concepts object and class	Corresponding dimensions of variation
Class is understood as an entity of the program, contributing to the structure of the code and describing the object.	The textual representation of the concepts.
As above, and in addition, class is understood as a description of properties and behaviour of objects, where object is understood as something that is active in the program.	As above, and in addition, the action of the program.
As above, and in addition, class is understood as a description of properties and behaviour of the object, where object is understood as a model of some real world phenomenon.	As above, and in addition, the modeling aspects of the concepts.

Table 6.2: *Categories describing the different understandings of the concepts* object *and* class*, and the corresponding dimensions of variation related to the critical features of the identified understandings.*

6.1.4 Implications for education - patterns of variation

Table 6.2 carries implications for teaching. Teaching is here defined in a wide sense, not restricted to lecturing, but may include for example programming assignments given to students, software tools introduced to students, lectures, Internet and fellow students, anything the students meet and choose to use in their learning. The whole organisation of the learning environment is in this sense teaching.

The educator can create learning conditions that enable students to discern new features of the concepts. In this context it means creating possibilities for experiencing variation in dimensions related to features. We know from the analysis of our data that *any* variation is not sufficient. By varying some things and keeping others invariant, we can create the conditions necessary for learning. As educators we know this can be done in several different ways, and yet it is a difficult task.

The claim that not any variation is sufficient for creating good learning conditions is important in computer programming and counter-intuitive to how we often teach. For example Kölling and Rosenberg (2001) write that novice students should read code, not only simple code but large programs including many classes which can help them understand what object-oriented programming is. The downside of this approach is that large programs often mean that a number of different features of several concepts are present and vary simultaneously. The present research, together with a number of classroom studies reported by Marton and Tsui (2004), indicate that if students are not introduced to the critical features in adequate ways, they may not discern these features, and simultaneous variation of several features may not always provide good learning conditions. This is evident in the following quotes from one of the students in the first investigation, when he or she discusses the contrast between learning mathematics and learning computer programming:

> Here [in the programming course] you feel as if you only learn a lot of examples. You know, we've gotten so many examples of everything, in some way it feels as if you don't understand the base from the beginning

All the examples have obviously not helped the student sufficiently since he or she says about the programming course:

> I think it has been difficult with concepts and stuff, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult

Marton et al. (2004, p. 16–17) discuss so called *patterns of variation* which are identified from empirical studies. The patterns are ways of systematically combining variation and invariance in the teaching. Four different patterns are identified that can be used by educators as a toolbox. The patterns are *contrast, generalization, separation,* and *fusion* respectively. In short the patterns means (quoting Paper IV):

contrast to contrast a phenomenon *P* to other related phenomena, to make it possible to discern *P* as a phenomenon distinct from other phenomena.

generalization to exhibit varying specific appearances of *P*, in order to open the possibility to discern the general meaning of *P*.

separation there is variation in precisely one dimension, to create the possibility to discern that particular dimension, keeping the other dimensions invariant.

fusion to exhibit variation in several dimensions simultaneously, to open the possibility to discern the relations between these dimensions.

Patterns of variation: some examples

The content of Table 6.2 can be implemented in the teaching and learning environment by use of patterns of variation in a number of ways. There is great freedom and possibility to adapt the results to the need and desire of each educator, study group and learning resources. The following paragraphs discuss possible ways to achieve this, by showing a few examples.

For the first category, the students need to become aware that different programs represent classes and objects differently, at a textual level. This corresponds to the second part of the first category. In the first part of the first category, the focus is on the structure of the program text. There are several aspects of a program structure. A single class has a structure in terms of its attributes and methods. Students also encounter problems including several classes where each class is an entity of the program. Both these aspects of the structure of the code need to be exposed in teaching. A way to achieve this is to use the *generalization* pattern, in a variety of simple UML class diagrams[2] (Rumbaugh et al., 1999). By exhibiting various specific appearances of classes, the general meaning of class and object as text can be discerned. To transfer the structure from the diagram to the code where the methods are separated from the attributes is possible even if there is only one single class and will show varying textual examples. This is often the case in the examples considered in the beginning of a programming course. The feature that the class is a help when structuring the program is made even more apparent when more than one class is used to solve a problem. Each class is represented in a UML diagram and forms its own entity of the program.

For the second category, the new feature focuses on actions during program execution. Different actions of the program taking place when the program is executed make a dimension of variation related to this feature. I will first discuss the *separation* pattern. The general idea of this pattern is that there is variation in precisely one dimension, so there is a possibility for the student to discern that particular dimension. This seems to be an appropriate pattern for our purpose. It is however difficult to achieve variation in one dimension only since a change in action requires a change in the program text. In Paper IV we suggest the notion of *pseudo separation*. In this context this means that the textual differences between two programs is kept small, but still causes a change in action when the program is executed. This will give the student the possibility of discerning the action dimension separately. Pseudo separation is a form of fusion pattern since in fact there is a variation in both the action dimension and the textual dimension, even though the latter is not prominent. When the student has discerned the action dimension, the proper fusion pattern can be used to show the *relation* between the program's textual representation and its actions. An example of a resource that can be used for the latter example is

[2]UML (Unified Modeling Language) is a visual language. It is a standard for modeling, developing and documenting object-oriented computer systems.

BlueJ (Barnes and Kölling, 2003), where a debugger can be used to execute the program in steps so the variation of the code and variation in values of variables can be observed simultaneously during program execution.

For the last category in Table 6.2 the feature added focuses on objects and classes as models of the real world. To help students discern the dimension related to the modeling feature, the *generalization* pattern can be used. Modeling appears in many areas in the students' lives. Road signs model real world phenomenon like road bumps and let us avoid long, written instructions. Mathematical symbols model complex relations like sums and integrals and simplify the treatment of computations. Modeling in computer programming helps us to treat complex real-world problems. Once the student has discerned the modeling dimension, the *fusion* pattern, where variations in several dimensions are exhibited simultaneously, can be used to help the student discern the relationship between the modeling dimension and the action and textual dimensions. Results from the first investigation point to the importance of letting students follow the whole process of a programming task, including the analysis of a problem in real life, and not only focus on implementing code. This can be implemented in teaching by using an assignment where several classes are needed. The first part of the assignment would be to do an object oriented analysis of a real world problem, deciding which classes are needed, which methods each class should include, and which information the classes need to exchange. If the students are in their first programming course, they may in a next step need help to modify their models to find suitable classes with attributes and methods before starting to code. After implementing and testing the code, the students are supposed to discuss in groups their different solutions, and how their final solutions differ from their first analysis. This might help the students to discern the real world feature of objects and classes, and also to discern the relationship between the real world problem, the model in terms of class diagrams, and the implementation of the problem as code.

For further examples on how results from phenomenographic analyses can be implemented in teaching, I refer to Paper IV where a phenomenographic analysis of novice students' understanding of what *computer programming* means is described. Critical features and corresponding dimensions of variation are identified, followed by a discussion on how patterns of variation can be used to open a space of learning for the students. In Paper V on the other hand, we present a phenomenographic analysis of novice students' understanding of what it means to *learn* computer programming. It has shown to be important for students to have a good understanding of what learning the subject means. The outcome space presented in Paper V can be used by educators in similar ways as the present discussion to facilitate for novice students to get a good foundation for their learning.

6.1.5 The results related to previous research

My results can shed new light upon and give explanation to other research and discussions in the field.

For example, Computing Curricula 2001 Section 7.2 (Roberts and Engel, 2001) writes:

> Introductory programming courses often oversimplify the programming process to make it accessible to beginning students, giving too little weight to design, analysis, and testing relative to the conceptually simpler process of coding. Thus, the superficial impression students take from their mastery of programming skills masks fundamental shortcomings that will limit their ability to adapt to different kinds of problem-solving contexts in the future.

This is in line with the discussion above on the need for students to follow a whole programming task, including the analysis to find suitable objects in a real world problem, to get a good understanding of object-oriented programming. Using terminology from variation theory, if the focus of introductory programming is on coding only, the textual dimension of variation is highlighted at the expense of the action and modeling dimensions.

As a second example I will discuss some misconceptions pointed to in the literature (Holland et al., 1997), namely an overemphasizing of the object's data feature at the expense of the behavioural feature and the "object as a kind of variable" misconception. The latter may occur if the examples students first come across have only one instance variable. Students with previous experience of procedural programming may develop the misconception that objects are in some sense mere wrappers for variables.

Both misconceptions point to the importance of understanding the concepts as they are described in the second categories in Table 6.2. The second category emphasizes that classes describe the behaviour of objects. The second category also explains classes as a description of properties of objects, and most real-world objects have more than one property.

Holland et. al give some advice on how to help students avoid these misconceptions. To increase the chances of avoiding the "object as a kind of variable" misconception the authors suggest that all the classes showed as an introduction should have more than one instance variable and that these variables should be of different type. Another way to avoid over-emphasising the object's data feature, suggested by the authors, is using introductory object examples where the response to a message is substantially altered depending on the state of the object. Holland et al.'s suggestions are in line with variation theory and the discussion in Section 6.1.4. Using variation theory terminology, examples with at least two instance variables of different types is using the generalization pattern, and examples where the response to a message substantially alters depending on the object is using the (pseudo) separation pattern as discussed above.

A third example, also mentioned by Holland et al. and Sanders and Thomas (2007), is the common problem among novice programmers of understanding the difference between class and object. This might become a problem if several examples are presented in which only a single instance of each class is used. Holland et al. suggest that it would help to avoid this misconception

if several instances of each class are always presented. As explained in Section 6.1.2, the textual representation of programs constitutes a dimension of variation. This implies variation in the sense of presenting more than one instance of the class in the code, as recommended by Holland et al., which is according to the first category in Table 6.2.

In the light of the present study, the recommendations from Holland et al. are explained by and theoretically rooted in variation theory. Variation theory and patterns of variation are thus scientifically based and empirically tested tools to be used by educators to develop their teaching.

As a fourth example, Holmboe (1999) performed a study where students who had just finished an introductory course on object-oriented programming, senior students, and educators, were asked to describe in their own words what object-oriented programming is. He made a qualitative analysis of the answers, and concludes that some types of knowledge are more suitable as a basis for further knowledge construction than others. He writes about the understanding that includes the world outside the computer itself: "A person with holistic knowledge relates the implementation and design of a computer program to the real world being simulated." Holmboe emphasizes the importance that "[...] more students will experience the connection between reality, model and implemented program, and thus reach holistic knowledge of object-orientation sooner in their learning process." The third category in Table 6.2 captures an understanding of classes and objects that includes the world outside the computer itself, the modeling of real-world phenomena, and Section 6.1.4 discusses how educators can facilitate for students to discern this understanding by use of patterns of variation.

One challenge for educators of object-oriented programming, is to construct an educational environment which facilitates for students to reach a rich understanding of the concepts object and class. To this end it is important to know the different ways in which students (as opposed to experts) typically experience these concepts. My phenomenographic study has given such insight. Next the educator needs to identify critical features and related dimensions of variation of the concepts the students need to discern in order to reach a rich understanding. Here, variation theory can be used, as demonstrated in the previous discussion. Finally, the patterns of variation are like a tool box for educators to open up dimensions of variation and thus give students opportunities to come to richer understandings of the concepts.

6.2 Dimensions of variation and student learning of practise

Paper VIII discusses students' learning of threshold concepts. The paper points to the important but problematic role of practise in programming students' learning, and how concepts and practise interact in the learning process.

Paper IX develops this line of research further. The paper gives examples of typical novice student programming activities related to the skills of reading, writing, and debugging code, see Table 6.3.

Read code: to discern main parts of short programs; to read code and recognize key words; to read code and understand what will happen when the instructions are executed; to read and relate code to the application and the problem domain.
Write code: to use an editor to emphasise the structured of a program by means of indents, empty lines etc.; to write common programming building blocks in a syntactically correct way; to design a short algorithm; to express a short algorithm in pseudo code; to implement pseudo code in a programming language; to design a solution to a whole problem and transfer the design to pseudo code, using common programming building blocks; to implement the solution to a problem according to basic software quality requirements.
Test and debug code: to use a compiler to find and correct minor syntax errors; to use the computer to execute code to verify expected output; to use a compiler to get executable code; to read and understand simple syntax errors, such as missing semicolon; to correct simple syntax errors, for example missing semicolon; to hand execute a program on paper before coding; to diagnose semantic errors in the code; to test code in relation to the problem domain and usability.

Table 6.3: *Common novice programming skills with associated activities.*

The paper discusses how the activities correspond to different levels of proficiency, and furthermore, how the activities relate to previously identified dimensions of variation. These dimensions were identified from the phenomenographic outcome space on novice students' understandings of the concepts object and class as discussed in Section 6.1.3. The identified dimensions of variation are thus related to different conceptual understandings as well as to activities at different levels of proficiency. In this way, the dimensions of variation act as interfaces between qualitatively different conceptual understandings and activities at different levels of proficiency.

There are implications for teaching following from these results. First, novices are often expected to perform many of the activities mentioned in Table 6.3 at an early stage of their education. Some of them are however related to dimensions of variation corresponding to a high level of proficiency. These dimensions are at the same time related to advanced ways of understanding concepts that we know very few of the students have discerned yet. This means, using variation theory terminology, that the students have not yet discerned the dimensions of variation related to the

activities, and we still expect them to manage them. This can to some extent explain why novice students have such big problems learning, and why the activities in the lab do not always lead to the expected learning outcome.

Another result from Paper IX is that to be able to discern a certain feature of a concept, *or* to make an activity meaningful, certain dimensions of variation in the learning space need to be open for the student. Or, to phrase the same thing differently: the learning of concepts *and* activities presupposes that related dimensions of variation are discerned. At the same time, the richer ways to see the concepts, and the activities at the higher level of proficiency, relate to more dimensions of variation and require thus that more dimensions *and* their relations be discerned.

Educators can use the results from Paper IX together with patterns of variation to facilitate students' learning, the conceptual as well as the practical. When dimensions of variation are identified, appropriate patterns of variation can be introduced to the students to facilitate the learning of corresponding concepts *and* practises.

7. Conclusions and future work

Computer programming is a core area in computer science education that involves practical and well as conceptual learning goals. It is however widely reported in the computer science education research literature that novice students have great problems in learning to program. The problems reported apply to both concepts and practise.

The research presented in this thesis contributes to the body of knowledge on students' learning by investigating the relationship between conceptual and practical learning in novice students' learning to program. Previous research in computer science education has focused either on students' learning practise or on concepts. The present research however indicates that students' problems with learning to program partly depend on a complex relationship and mutual dependence between the two.

The most common way to reach practical as well as conceptual learning goals in programming education is to "learn through practising". Students are expected to "learn to do the practise" as well as to learn the concepts through practising. If the students do not master the practise this might hinder further learning, conceptual as well as practical. The present research indicates that the students find practise at least as difficult to learn as concepts. The practise in not merely the unproblematic means of reaching the learning goals.

The research builds on three empirical investigations. The data from the investigations have been analysed from several perspectives. Students' conceptual and practical learning are first investigated separately by means of content analysis, and phenomenography and variation theory.

The analyses of student learning of concepts show that many students have problems to learn central concepts. The analyses show however how phenomenographic results can be used to facilitate for students' learning by use of variation theory. The analysis of students' ability to master the practise shows that students hold a great variety of strategies that they can use when they are stuck in their learning. On the other hand senior computer science students perform poorly when asked to perform a design tasks. Design is a core skill in computer science education. There are obviously problems in students' achievements of the practical learning goal.

In a subsequent analysis inspired by phenomenography and variation theory I show that practise, in terms of programming activities at different levels of proficiency, as well as conceptual understandings at qualitatively different levels, are related to dimensions of variation.

Previous phenomenographic research points to how critical features of concepts are related to dimensions of variation. Previous research also suggests

that practise can be related to dimensions of variation. The most significant finding in the present thesis is that I have demonstrated that practise, in terms of activities at different level of proficiency, and qualitatively different conceptual understandings, have dimensions of variation *in common*. This has been possible since I propose a way to identify dimensions of variation related to practises.

An analytical model is suggested where the dimensions of variation are like interfaces, relating concepts and activities. The implications of the model are several. If the dimensions of variation are at the center of the learning process this implies that when students discern a dimension of variation, related conceptual understandings *and* the meaning embedded in related practises can be discerned.

The model further suggests that activities as well as concepts can relate to more than one dimension. This implies that activities at a higher level of proficiency, as well as qualitatively richer understandings of concepts, relate to more dimensions of variation. The analysis on novice students' conceptual understandings points to dimensions of variation that many of the novice students not seem to have discerned. The analysis of students' activities shows that some of the activities students' often are expected to do and learn early in their education, relate to these dimensions of variation that the former study showed were problematic to discern. This can to some extent explain why the exercises in the computer lab do not always lead to improved learning. The results can furthermore be used by educators to help students' discern dimensions of variation and thus facilitate for the learning, practical as well as conceptual. A concrete example is given on how variation theory and patterns of variation can be applied in programming education.

The results need further investigations. Phenomenography and variation theory (Marton and Booth, 1997; Marton and Tsui, 2004) traditionally discuss ways to identify critical features of phenomena like concepts, and ways to open a space of learning for students by means of patterns of variation in the teaching. The present work contributes to the body of knowledge of the student learning by proposing a way to identify dimensions of variation related to practise. Furthermore, the research proposes a model which demonstrates how dimensions of variation are like interfaces between concepts and practise, and between several concepts and several practises. There is a need of further empirical studies on how practise relates to dimensions of variation, and on the relationship between conceptual and practical learning. The analytical model can thus be used. This line of research might be possible by performing Learning Studies (Lo et al., 2004) which focus on educationally critical features of concepts and related practises.

Summary in Swedish

Nybörjarstudenters lärande av begrepp och praktik i programmering

Programmering är ett kärnämne inom datavetenskapliga utbildningar på universitetsnivå. Undervisning i programmering har lärandemål som gäller praktik lika väl som begrepp. Forskning i datavetenskapens didaktik visar emellertid att nybörjarstudenter har stora svårigheter att lära sig programmering. De rapporterade svårigheterna gäller såväl praktik som begreppsförståelse.

Forskningen i den här avhandlingen bidrar till befintlig forskning genom att undersöka relationen mellan begreppsligt och praktiskt lärande med fokus på nybörjarstudenters lärande av objekt-orienterad programmering. Tidigare forskning inom datavetenskapens didaktik har antingen fokuserat på studenters lärande av praktik, eller på lärandet av begrepp. Trots många försök att utveckla undervisningen kvarstår problemen med nybörjarstudenters lärande. Avhandlingen visar emellertid att studenters problem att lära sig programmering delvis beror på ett komplext samspel mellan och ett ömsesidigt beroende av praktik och begrepp i lärandeprocessen.

Det vanligaste sättet att nå de praktiska såväl som de begreppsliga lärandemålen i programmeringsutbildningar är att "lära genom att göra praktik", det vill säga genom att skriva datorprogram. Studenternas lärande av de praktiska såväl som de begreppsliga lärandemålen beror till stor del på om de klarar av att "göra praktik". Avhandlingen pekar på att studenterna erfar praktiken åtminstone lika svår att lära som koncepten. Praktiken är inte bara ett oproblematiskt medel att nå de konceptuella lärandemålen.

Forskningen bygger på tre empiriska studier. Den första studien undersökte nybörjarstudenters förståelse av några centrala begrepp inom objekt-orienterad programmering. 14 civilingenjörsstudenter inom området miljö- och vattenteknik intervjuades. Data från den första studien har främst analyserats med en fenomenografisk och variationsteoretisk forskningsansats.

Den andra studien fokuserade på att identifiera centrala och för studenterna problematiska begrepp, så kallade tröskelbegrepp. 16 sistaårsstudenter med datavetenskaplig inriktning intervjuades. Data från den andra studien har analyserats med en innehållsanalytisk forskningsansats.

Syftet med den tredje studien var att undersöka om studenter i slutet av sin datavetenskapliga utbildning kan designa datorprogram. Data från undersökningarna, designer gjorda av studenterna under kontrollerade former, analyserades med en innehållsanalytisk forskningsansats.

Studenternas lärande av begrepp och praktik analyserades först var för sig. Därefter undersöktes relationen mellan begreppsligt och praktiskt lärande.

Den fenomenografiska analysen av nybörjarstudenternas begreppsförståelse visar kvalitativt skilda sätt på vilka studenterna förstår, eller uppfattar, några centrala begrepp i objekt-orienterad programmering. Resultatet tyder på att många studenter har problem att lära sig de mer avancerade sätten att förstå begreppen, som också är de önskvärda från ett utbildningsperspektiv. Den variationsteoretiska analysen visar emellertid att variationsmönster (eng. *patterns of variation*) kan användas av lärare på resultat från den fenomenografiska analysen för att stödja studenter i deras lärande.

Analysen av hur studenter klarar de praktiska lärandemålen visar att studenterna besitter en stor variation av strategier som de kan använda när de får problem i sina studier. Studien visar också att studenter i slutet av sin datavetenskapliga utbildning presterade sämre än förväntat på designuppgiften. Design är ett kärnområde i datavetenskaplig utbildning som studenterna förväntas lära sig. Resultaten av analysen tyder på att det finns problem med studenters förvärvande av de praktiska lärandemålen.

Analysen av relationen mellan begreppsligt och praktiskt lärande är inspirerad av fenomenografi och variationsteori. Den visar att såväl praktiken, i termer av programmeringsaktiviteter på olika färdighetsnivåer, som kvalitativt skilda förståelser av begrepp, är relaterade till variationsdimensioner (eng. *dimensions of variation*).

Tidigare fenomenografisk forskning pekar på hur kritiska aspekter av begrepp är relaterade till variationsdimensioner. Tidigare forskning föreslår också att praktik kan relateras till variationsdimensioner. Det mest signifikanta resultatet i avhandlingen är att det visas att praktiken, i termer av programmeringsaktiviteter på olika färdighetsnivåer, och kvalitativt skilda begreppsliga förståelser, har *gemensamma* variationsdimensioner. Avhandlingen beskriver ett sätt att relatera variationsdimensioner till programmeringsaktiviteter, vilket har gjort det möjligt att komma fram till resultatet att begrepp och aktiviteter kan ha gemensamma variationsdimensioner.

En analytisk modell föreslås där variationsdimensioner fungerar som gränssnitt mellan begrepp och aktiviteter. Modellen har flera implikationer. Om variationsdimensioner är i centrum av lärandeprocessen, innebär det att när studenter urskiljer en variationsdimension, kan relaterade begreppsliga förståelser *och* meningen i relaterade aktiviteter urskiljas.

Modellen visar dessutom att såväl aktiviteter som begrepp kan relatera till mer än en variationsdimension. En tolkning av det resultatet är att aktiviteter på en högre färdighetsnivå, likaväl som kvalitativt rikare begreppsliga förståelser, relaterar till fler variationsdimensioner. Analysen av studenternas begreppsliga förståelser pekar på att många inte har uppfattat alla variationsdimensioner. Analysen visar ytterligare att vissa aktiviteter som studenterna förväntas kunna på ett tidigt stadium av sin utbildning, relaterar till just de variationsdimensioner som den tidigare nämnda studien pekar

på som svåra att uppfatta. Det nämnda resultatet kan till viss utsträckning förklara varför de praktiska övningarna i programmeringsundervisningen inte alltid leder till ökat lärande, varken av begrepp eller praktik.

Det är viktigt för läraren att kunna identifiera variationsdimensioner så att dessa kan lyftas fram i undervisningen. Avhandlingen ger konkreta exempel på hur variationsteori och variationsmönster kan användas i programmeringsundervisning. Utgående från ett fenomenografiskt utfallsrum visas hur kritiska aspekter av de olika förståelserna beskrivna i utfallsrummet kan identifieras. Varje kritisk aspekt relaterar till en variationsdimension. Därefter diskuteras hur olika variationsmönster kan användas för att lyfta fram variationsdimensioner i undervisningen, vilket kan hjälpa studenter att urskilja de identifierade variationsdimensionerna. Läraren kan på så sätt ge möjlighet till studenterna att lära sig relaterade begrepp *och* praktik på nya sätt.

Acknowledgments

At the end of my PhD studies I look back on a joyful but also demanding journey. Many people have helped and encouraged me during the journey. There are friends and relatives, colleagues and students, who have been supporters and contributed to the thesis, and I would like to thank them all. I will mention some of them below.

First of all I would like to thank Michael Thuné who has been my supervisor during the whole thesis work, and Anders Berglund who has been my supervisor after my licentiate thesis, for your great support during the process of creating this thesis. Your knowledge in the two areas my research spans, your advice, patience, and encouragement to me to try my own ideas, and our exciting discussions and collaboration on papers have been invaluable in my thesis work. I would also like to thank Shirley Booth who introduced me to the phenomenographic research approach during my licentiate work.

I would also like to thank my co-authors of papers in the thesis, the Sweden Group, for the enjoyable and rewarding research we have performed together (in alphabetic order): Jonas Boustedt, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, Lynda Thomas, and Carol Zander. I would especially like to thank Robert McCartney and Kate Sanders for their English proof reading the thesis, and Jonas Boustedt and Carol Zander for enjoyable travel to conferences and research meetings, and for long and rewarding discussions.

I also want to thank my research group at the Department of Information Technology at Uppsala University for valuable seminars with discussions, feedback, and encouragement, and all colleagues at the Department of Scientific Computing for a warm and stimulating work environment. Specifically I want to thank Liselott Dominicus for being a friend and traveling companion on the journey to the PhD.

Finally I would like to thank my family, Per, Nils, and Olof, who have made everything worthwhile. In this I also include my mother Anne-Mari Sundin who has encouraged me to start by saying:

> *Bättre lyss till den sträng som brast än aldrig spänna sin båge.*

and has continued to support my work throughout.

My thesis work has been financed by The Swedish Research Council, and Faculty of Educational Sciences, Uppsala University.

Bibliography

ACM Curriculum Committee on Computer Science (1968). Curriculum '68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 11(3):151–197.

Austing, R. H., Barnes, B. H., and Engel, G. L. (1977). A survey of the literature in computer science education since curriculum '68. *Communications of the ACM*, 20(1):13–21.

Barnes, D. and Kölling, M. (2003). *Objects First with Java - A Practical Introduction using BlueJ*. Prentice Hall/Pearson Education.

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

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional.

Ben-Ari, M. (1998). Constructivism in computer science education. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 257–261, New York, NY, USA. ACM.

Berglund, A. (2005). *Learning computer systems in a distributed project course. The what, why, how and where*. Number 62 in Uppsala Dissertations from the Faculty of Science and Technology. Acta Universitatis Upsaliensis,
Uppsala, Sweden.

Berglund, A., Daniels, M., and Pears, A. (2006). Qualitative research projects in computing education research: an overview. In *ACE '06: Proceedings of the 8th Austalian conference on Computing education*, pages 25–33. Australian Computer Society.

Booth, S. A. (1992). *Learning to Program. A phenomenographic perspective*. Number 89 in Göteborg Studies in Educational Science. Acta Universitatis Gothoburgensis, Göteborg, Sweden.

Boustedt, J. (2007). *Students Working with a Large Software System: Experiences and Understandings*. Licentiate thesis, Uppsala University, Uppsala, Sweden.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? *SIGCSE Bulletin*, 39(1):504–508.

Bruce, C., McMahon, C., Buckingham, L., Hynd, J., Roggenkamp, M., and 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.

Clancy, M., Stasko, J., Guzdial, M., Fincher, S., and Dale, N. (2001). Models and Areas for CS Education Research. *Computer Science Education*, 11(4):323–341.

Collier Reed, B., Ingerman, Å., and Berglund, A. (2008). Reflections on trustworthiness in phenomenographic research:ărecognising purpose, context and change in the process of research. *Education as Change*, (in press).

Daly, C. and Waldron, J. (2004). Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213.

Davies, S. P. (1993). Models and theories of programming strategy. *International journal of Man-Machine Studies*, 39(2):237–267.

Denzin, N. K. and Lincoln, Y. S. (1994). Introduction. Entering the Field of Qualitative Research. In Denzin, N. K. and Lincoln, Y. S., editors, *Handbook of Qualitative Research*, pages 1–17. SAGE Publications.

Denzin, N. K. and Lincoln, Y. S. (2005). Introduction: The discipline and practice of qualitative research. In Denzin, N. K. and Lincoln, Y. S., editors, *The SAGE Handbook of Qualitative Research third edition*, pages 1–32. SAGE Publications.

Eckerdal, A. (2006). *Novice Students' Learning of Object-Orientd Programming*. Licentiate thesis, Uppsala University, Uppsala, Sweden.

Eckerdal, A. (2009). Ways of Thinking and Practising in Introductory Programming. Technical Report 2009-002, Department of Information Technology, Uppsala University, Sweden.

Eckerdal, A. and Berglund, A. (2005). What Does It Take to Learn 'Programming Thinking'? In *Proceedings of the 1st International Computing Education Research Workshop, ICER*, pages 135–143, Seattle, Washington, USA.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2006a). Putting threshold concepts into context in computer science education. *SIGCSE Bulletin*, 38(3):103–107.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Zander, C. (2006b). Can graduating students design software systems? In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 403–407.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliff, M., and Zander, C. (2006c). Categorizing student software designs: Methods, results, and implications. *Computer Science Education*, 16(3).

Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., Thomas, L., and Zander, C. (2007). From Limen to Lumen: Computing students in liminal spaces. In *Proceedings of the 3rd International Workshop on Computing Education Research*, pages 123–132. ACM.

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

Ellis, A., Carswell, L., A., B., Deveaux, D., Frison, P., Meisalo, V., Meyer, J., Nulden, U., Rugelj, J., and Tarhio, J. (1998). Resources, tools, and techniques for problem based learning in computing. In *ITiCSE-WGR '98: Working Group reports of the 3rd annual SIGCSE/SIGCUE ITiCSE conference on Integrating technology into computer science education*, pages 41–56.

Entwistle, N. (2003). Concepts and conceptual frameworks underpinning the ETL project. Occasional Report 3 of the Enhancing Teaching-Learning Environments in Undergraduate Courses Project, School of Education, University of Edinburgh, March 2003.

Entwistle, N. (2007). Conceptions of learning and the experience of understanding: Thresholds, contextual influences, and knowledge objects. In Vosniadou, S., Baltas, A., and Vamvakoussi, X., editors, *Re-framing the Conceptual Change Approach in Learning and Instruction*, pages 123–143. ELSEVIER.

Fazey, J. and Marton, F. (2002). Understanding the space of experiential variation. *Active Learning in Higher Education*, 3(3):234–250.

Fincher, S. and Petre, M. (2004). Mapping the territory. In Fincher, S. and Petre, M., editors, *Computer Science Education Research*, pages 1–8. Routledge.

Finlay, L. (2002). "Outing" the Researcher: The Provenance, Process, and Practice of Reflexivity. *Qualitative Health Research*, 12(4):531–545.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116.

Fleury, A. E. (1999). Student conceptions of object-oriented programming in Java. *The Journal of Computing in Small Colleges*, 15(1):69–78.

Fleury, A. E. (2000). Programming in Java: Student-Constructed Rules. In *Proceedings of the 31st SIGCSE technical symposium on Computer science education*, pages 197–201, Austin, Texas, United States.

Fleury, A. E. (2001). Encapsulation and Reuse as Viewed by Java Students. *ACM SIGCSE Bulletin*, 33(1):189–193.

Fung, P., Brayshaw, M., and du Boulay, B. (1990). Towards a taxonomy of novices' misconceptions about the prolog interpreter. *Instructional Science*, 19(4-5):311–336.

Goldman, K. J. (2004). A concepts-first introduction to computer science. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 432–436. ACM.

Goldweber, M., Clark, M., and Fincher, S. (2004). The relationship between CS education research and the SIGCSE community. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 147–148. ACM.

Graneheim, U. and Lundman, B. (2004). Qualitative content analysis in nursing research: concepts, procedures and measures to achieve trustworthiness. *Nurse Education Today*, 24(2):105–112.

Gross, P. and Powers, K. (2005). Evaluating assessments of novice programming environments. In *Proceedings of the 1st International Computing Education Research Workshop, ICER, Seattle, Washington, USA*, pages 99–110.

Gugerty, L. and Olson, G. (1986). Debugging by skilled and novice programmers. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 171–174, New York, NY, USA. ACM.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of computer science. *Computer Science Education*, 13(2):95–122.

Hoepfl, M. C. (1997). Choosing Qualitative Research: A Primer for Technology Education Researchers. *Journal of Technology Education*, 9(1):47–63.

Holland, S., Griffiths, R., and Woodman, M. (1997). Avoiding object misconceptions. In *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 131–134.

Holmboe, C. A. (1999). A cognitive framework for knowledge in informatics: The case of object-orientation. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 17–20.

Hsieh, H.-F. and Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative Health Research*, 15(9):1277–1288.

Kahney, H. (1983). What do novice programmers know about recursion. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 235–239, New York, NY, USA. ACM.

Katira, N., Williams, L., Wiebe, E., Miller, C., Balik, S., and Gehringer, E. (2004). On understanding compatibility of student pair programmers. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 7–11.

Kirk, J. and Miller, M. L. (1986). *Reliability and validity in qualitative research*. Sage Publications.

Kölling, M. (1999a). The problem of teaching object-oriented programming, part 1: Languages. *JOURNAL OF OBJECT-ORIENTED PROGRAMMING*, 11(8):8–15.

Kölling, M. (1999b). The problem of teaching object-oriented programming, part 2: Environmentss. *JOURNAL OF OBJECT-ORIENTED PROGRAMMING*, 11(9):6–12.

Kölling, M. and Rosenberg, J. (2001). Guidelines for teaching object orientation with java. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36, New York, NY, USA. ACM.

Krippendorff, K. (2004). *Content Analysis: An Introduction to Its Methodology*. Thousand Oaks, Calif.: Sage.

Lincoln, Y. S. and Guba, E. G. (1985). *Naturalistic Inquiry*. SAGE Publications.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150.

Lo, M. L., Marton, F., Pang, M. F., and Pong, W. Y. (2004). Toward a pedagogy of learning. In Marton, F. and Tsui, A., editors, *Classroom Discourse and the Space of Learning*, pages 189–225. Lawrence Erlbaum Associates, Mahwah, NJ.

Lopez, M., Whalley, J., and Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the forth International Computing Education Research Workshop*, pages 101–111.

Marton, F. and Booth, S. (1997). *Learning and Awareness*. Lawrence Erlbaum Ass., Mahwah, NJ.

Marton, F., Hounsell, D., and Entwistle, N. (1984). *The Experience of Learning*. Scottish Academic Press.

Marton, F., Runesson, U., and Tsui, A. (2004). The space of learning. In Marton, F. and Tsui, A., editors, *Classroom Discourse and the Space of Learning*, pages 3–40. Lawrence Erlbaum Ass., Mahwah, NJ.

Marton, F. and Svensson, L. (1979). Conceptions of research in student learning. *Higher Education*, pages 471–486.

Marton, F. and Tsui, A. (2004). *Classroom Discourse and the Space of Learning*. Lawrence Erlbaum Ass., Mahwah, NJ.

Mayring, P. (2000). Qualitative content analysis. Forum: Qualitative Social Research [On-line Journal], 2000, 1(2) Available at: http://www.qualitative-research.net/fqs-texte/2-00/2-00mayring-e.htm.

McCartney, R., Eckerdal, A., Mostrom, J. E., Sanders, K., and Zander, C. (2007). Successful students' strategies for getting unstuck. *SIGCSE Bulletin*, 39(3):156–160.

McCormick, R. (1997). Conceptual and procedural knowledge. *International Journal of Technology and Design Education*, 7(1–2):141–159.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.-D., 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. *SIGCSE Bulletin*, 33(4):125–180.

McCune, V. and Hounsell, D. (2005). The development of students' ways of thinking and practising in three final-year biology courses. *Higher Education*, 49:255–289.

Meyer, B. (1988). *Object-oriented Software Construction*. International series in Computer Science. Prentice Hall.

Meyer, J. H. and Land, R. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):373–388.

Molander, B. (1996). *Kunskap i handling*. DAIDALOS.

Molander, B., Halldén, O., and Pedersen, S. (2001). Understanding a Phenomenon in Two Domains as a Result of Contextualization. *Scandinavian Journal of Educational Research*, 45(2):115–123.

Mostyn, B. (1985). The Content Analysis of Qualitative Research Data: A Dynamic Approach. In Brenner, M., Brown, J., and Canter, D., editors, *THE RESEARCH INTERVIEW Uses and Approaches*, pages 115–145. ACADEMIC PRESS INC. (LONDON) LTD.

Newman, I., Daniels, M., and Faulkner, X. (2003). Open ended group projects, a 'tool' for more effective teaching. In *Proceedings of the fifth Australasian conference on Computing education*, pages 95–103.

Pears, A., Seidman, S., Eney, C., Kinnunen, P., and Malmi, L. (2005). Constructing a core literature for computing education research. *ACM SIGCSE Bulletin*, 37(4):152–161.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223, New York, NY, USA. ACM.

Posner, G., Strike, K., Hewson, P., and Gertzog, W. (1982). Accommodation of a scientific conception: toward a theory of conceptual change. *Science Education*, 66(2):211–227.

Powers, K., Cooper, S., Goldman, K., Carlisle, M., McNally, M., and Proulx, V. (2006). Tools for teaching introductory programming: What works? In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 560–561.

Powers, K., Ecott, S., and Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 213–217, New York, NY, USA. ACM.

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

Randolph, J. (2007). *Computer science education research at the crossroads: A methodological review of the computer science education research: 2000-2005*. PhD dissertation: Utah State University http://www.archive.org/details/randolph_dissertation Retrieved November 19, 2008.

Roberts, E. and Engel, G. (2001). Computing Curricula 2001: Final Report of the Joint ACM/IEEE-CS Task Force on Computer Science Education. IEEE Computer Society Press, December 2001, http://www.acm.org/sigcse/cc2001/.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Reading, Massachusetts.

Runesson, U. (2006). What is it Possible to Learn? On Variation as a Necessary Condition for Learning. *Scandinavian Journal of Educational Research*, 50(4):397–410.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., , Thomas, L., and Zander, C. (2008). Student understanding of object-oriented programming as expressed in concept maps. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 332–336.

Sanders, K., Fincher, S., Bouvier, D., Lewandowski, G., Morrison, B., Murphy, L., Petre, M., Richards, B., Tenenberg, J., Thomas, L., Anderson, R., Anderson, R., Fitzgerald, S., Gutschow, A., Haller, S., Lister, R., McCauley, R., McTaggart, J., Prasad, C., and Scott, T. (2005). A multi-institutional, multinational study of programming concepts using card sort data. *Expert Systems*, 22(3):121–128.

Sanders, K. and Thomas, L. (2007). Checklists for grading object-oriented cs1 programs: concepts and misconceptions. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 166–170, New York, NY, USA. ACM.

Simon (2007). A Classification of Recent Australasian Computing Edcuation Publications. *Computer Science Education*, 17(3):155–169.

Spohrer, J. C. and Soloway, E. (1986). Alternatives to construct-based programming misconceptions. *SIGCHI Bulletin*, 17(4):183–191.

Stamouli, I. and Huggard, M. (2006). Object Oriented Programming and Program Correctness: The Students' Perspective. In *ICER '06: Proceedings of the second International Workshop on Computing Education Research*, pages 109–118, Canterbury, United Kingdom.

Séré, M. (2002). Towards renewed research questions from the outcomes of the european project *Labwork in Science Education*. *Science Education*, 86(5):624–644.

Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R., Monge, A., Moström, J., Petre, M., Powers, K., Ratcliffe, M., Robins, A., Sanders, D., Shwartzman, L., Simon, B., Stoker, C., Tew, A., and VanDeGrift, T. (2005). Students designing software: a multinational, multi-institutional study. *Informatics in Education*, 4(1):143–162.

Thuné, M. and Eckerdal, A. (2009). Variation Theory Applied to Students' Conceptions of Computer Programming. *European Journal of Engineering Education*, Accepted for publication.

Trigwell, K., Prosser, M., and Taylor, P. (1994). Qualitative differences in approaches to teaching first year university science. *Higher Education*, 27(1):75–84.

Valentine, D. (2004). CS educational research: a meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE Bulletin*, 36(1):255–259.

VanDeGrift, T. (2004). Coupling pair programming and writing: Learning about students' perceptions and processes. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 2–6.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K., and Prasad, C. (2006). An austalasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *ACE '06: PProceedings of the 8th Australian conference on Computing education*, pages 243–252.

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

Zou, L. and Godfrey, M. W. (2008). Understanding interaction differences between newcomer and expert programmers. In *RSSE '08: Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 26–29, New York, NY, USA. ACM.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 600

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-9551

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2009

Paper I

Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory

Anna Eckerdal
Department of Information Technology
Uppsala University
P.O. Box 337, 751 05 Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Michael Thuné
Department of Information Technology
Uppsala University
P.O. Box 337, 751 05 Uppsala, Sweden
Michael.Thune@it.uu.se

ABSTRACT

Problems with understanding concepts, so called misconceptions, have been investigated and reported in a number of studies regarding object-oriented programming [4], [3]. In a first programming course using an object-oriented language, it is of great importance that students get a good understanding of central concepts like *object* and *class* at an early stage of their education. We have, with a phenomenographic research approach, performed a study with first year university students, investigating what an understanding of the concepts *object* and *class* includes from a student perspective. By applying variation theory [8] to our results we are able to pin-point what the students need to be able to discern in order to gain a "rich" understanding of these concepts.

Categories and Subject Descriptors

K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Computer science education*; D.1.5 [**PROGRAMMING TECHNIQUES**]: Object-oriented Programming—*Java*
; D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*Cla-sses and objects*

General Terms

Human Factors, Theory

Keywords

Conceptions, misconceptions, phenomenography, variation theory

1. INTRODUCTION

Java is an often used first programming language in introductory programming courses for university students. There are many reports on problems with teaching Java [6], pointing out difficulties to understand central concepts in the object-oriented paradigm [3]. The study reported in this paper has a focus on students' different understandings of some central concepts in object-oriented programming. We identify different understandings of the concepts expressed by the students in the group. These understandings are critical from the students' perspective [10]. They also cover most of an expert understanding [9]. We claim that it is possible to establish general guidelines on how to organize the teaching and learning environment in such a way that students can get a good understanding of the concepts in question, and thus avoid misconceptions. We first give a theoretical background for the study and the analysis performed. The study and the results will then be presented, and after that we discuss implications for teaching following from the results from the study. The general implications for teaching are well in line with the results from other studies, and give a theoretical basis for explaining these results and how to generalize them.

2. PREVIOUS RESEARCH

There are many studies on *mis*conceptions of object-oriented concepts. The studies by Fleury [3] and Holland, Griffiths, and Woodman [4] are good example of this line of research. Fleury performed a study on student-constructed rules in beginning programming courses, where Java was taught, pointing out misconceptions among students. Holland, Griffiths and Woodman reported on misconceptions observed among students in a distance course where Smalltalk was taught. Our study differs from these, since it focuses on students' *conceptions* rather than misconceptions. We have found very few studies of this kind in the literature on how students learn to program. Fleury's constructivistically based study of students' understandings of objet-oriented programming [2], is an example. Another is Booth's phenomenographical investigation of how students' experience functional programming [1]. Our study, as Fleury's, has its focus on object-oriented concepts. However, like Booth, we use a phenomenographic approach (see below). The rationale for considering conceptions instead of misconceptions is the following. The conceptions found among students typically correspond to different aspects of a correct understanding of the concepts of interest. Since these are the conceptions actually formed by the students, they reveal ways to understand the concepts, that are of decisive importance from the students' perspective. We argue that

ITiCSE'05, June 27–29, 2005, Monte de Caparica, Portugal.

focusing on these crucial aspects in the teaching can help the students to gain a good understanding, and thus avoid different kinds of misconceptions.

3. RESEARCH APPROACH

3.1 Phenomenography

Variation theory is a tool to support and give guidelines in different educational settings. Variation theory has developed from Phenomenography [8]. Phenomenography aims at describing the variation of understandings of a certain phenomenon found in a group of people. Marton and Booth discuss the idea of phenomenography:

> The unit of phenomenographic research is *a way of experiencing something,* [...], and the object of the research is the *variation* in ways of experiencing phenomena. At the root of phenomenography lies an interest in describing the phenomena in the world as other see them, and in revealing and describing the variation therein, especially in an educational context [...]. This implies an interest in the variation and change in capabilities for experiencing the world, or rather in capabilities for experiencing particular phenomena in the world in certain ways. These capabilities can, as a rule, be hierarchically ordered. Some capabilities can, from a point of view adopted in each case, be seen as more advanced, more complex, or more powerful than other capabilities. Differences between them are educationally critical differences, and changes between them we consider to be the most important kind of learning. [8, p. 111]

And later:

> [...] the *variation* in ways people experience phenomena in their world is a prime interest for phenomenographic studies, and phenomenographers aim to describe that variation. They seek the totality of ways in which people experience, or are capable of experiencing, the object of interest and interpret it in terms of distinctly different categories that capture the essence of the variation, a set of categories of description [...] [8, p. 121-122]

The object of interest in a phenomenographic study is thus how a certain phenomenon is experienced by a certain group of people. A fundamental assumption in phenomenography is that there exists only a limited number of qualitatively different ways in which a certain phenomenon can be understood.

Phenomenography is an empirical, qualitative research approach. It is often used in educational settings. Data can, like in the present study, be gathered in the form of interviews. The interviews are transcribed and analysed. Researchers, often more than one, analyse the data in order to find qualitatively different ways to understand the phenomenon expressed in the data. The researcher formulates the essence of the understandings found with his or her own words as categories of description. It is important to state that the analysis is on a group level, not aiming at presenting individual students' understandings, but the different understandings found in the group. This is done by reading and rereading the interviews, in context, but also by decontextualising excerpts and comparing them and grouping them together in different categories of understandings. The resulting description of qualitatively different categories of understanding constitutes the *outcome space* of the phenomenographic analysis.

3.2 Variation Theory

According to the phenomenographic tradition, the learning process is a question of *discerning* new aspects of phenomena. A specific aspect cannot however be discerned without experiencing *variation* in a "dimension" corresponding to that aspect. These dimensions are characteristic for the specific aspects, and the variations make central features of these aspects visible [10, p. 146].

With the phenomenographic outcome space as the starting point, the results can be applied in education, by using variation theory. When the empirical data is studied it is possible to discern the focus of each understanding expressed in the categories of description. In the phenomenographic analysis we identify aspects of the understanding of the phenomenon, critical for the understanding from the students' perspective. Learning requires discernment of new aspects of the phenomenon, and the teacher can create the conditions for such discernment with the judicious use of variation. By varying examples and problems and holding the critical aspect of the phenomenon invariant, that critical aspect is lifted out of the surrounding "noise". We speak of opening a dimension of variation, in which taken-for-granted ways of understanding are now brought into focus. Identifying these dimensions of variation corresponding to the critical aspects, gives a basis for finding implications for teaching.

4. THE STUDY

4.1 The Interviews

A study has been performed where 14 first year university students were interviewed on their understandings of the concepts *object* and *class*. The students had just finished their first programming course, a compulsory course giving 4 credit points. (At Swedish universities one credit point represents one week's full-time study.) The programming language used in the course was Java.

The interviews were semi-structured [7] with the aim to encourage the students to demonstrate as much as possible of their understandings and experiences within the theme of the interview. The interviewer had prepared a small number of questions, intended to approach the phenomena of object and class in different ways, to give the opportunity for the students to express as much of their understanding as possible.

Let us emphasize once again that our study is qualitative. Thus we are not aiming for statistically significant results. The objective in selecting persons to interview was to get as broad a coverage as possible of different conceptions. For that reason, most students taking part in the course filled in a questionnaire about previous programming knowledge, education, work experiences and gender. On the basis of these answers, we selected interviewees that represented as broad a coverage as possible of the factors mentioned.

4.2 The Phenomenographic Analysis

The interviews were transcribed and analysed. Two researchers independently read and analysed the interviews, looking for qualitatively different ways to understand the concepts *object* and *class* expressed in the data. Our results were very similar. We agreed upon three different ways to understand the concepts found in the data.

The different understandings of the concepts *object* and *class* found in the data are presented in Table 1 and Table 2

respectively. The understandings are inclusive. This means that an understanding expressed in one of the latter categories includes the understandings expressed in the former categories. Below, the categories in Table 1 and Table 2 are illustrated by excerpts from interviews. In the quotes, the interviewer is labeled I, and the students A, B, C etc.

4.2.1 The Concept of "object"

The different comprehensions of the concept *object* found in this study, can be formulated in three categories of description presented in Table 1.

Object is experienced as a piece of code.
As above, and in addition object is experienced as something that is active in the program.
As above, and in addition object is experienced as a model of some real world phenomenon.

Table 1: Categories describing the different ways to understand the phenomenon *object* found in the group.

In the first category, the understanding of the concept is limited to focus on the code as text. Student C says about objects:

> I imagine that it is a piece of code with all the variables piled under

When the interviewer asks the student how he/she would explain to a friend, who does not know anything about programming, what an object is, student N answers:

> I'd just say that it is a part of the program.

In the second category the comprehension is extended to include the results of the program execution, and the task of the object. It can be illustrated by the following answers.
Student H says:

> the object is a kind of, what is doing something [..] because it is all about that something is going to happen.

Student J says:

> If you think of the Java program, that it is built of different objects and it is the objects we modify so that we can get what we want from it.

The third category describes an understanding that an object is a model of some real world phenomenon. This is expressed in the following quotes:

> C: Yes an object, you can have a rather physical image of it....
> I: What did you say, physical?
> C: Kind of, you can think of a car and then it has one variable for how many wheels it has, one variable for the size of the engine like that.

The three categories express an increasing complexity. The first category shows an understanding that all students express in one way or the other, objects as they appear in the code. A few students express only this understanding. This category expresses a poor understanding, while the last one shows a rich understanding including fundamental ideas behind the object-oriented paradigm.

4.2.2 The Concept of "class"

When looking for the different understandings of the concept *class* expressed in the study, a pattern similar to the understandings of the concept *object* is found. There are comprehensions focusing on the code and the task of the programmer, but there are also comprehensions where the reality the program is supposed to model is present. The categories of description are presented in Table 2 and illustrated by quotes below.

Class is experienced as an entity in the program, contributing to the structure of the code.
As above, and in addition class is experienced as a description of properties and behaviour of the object.
As above, and in addition class is experienced as a description of properties and behaviour of the object, as a model of some real world phenomenon.

Table 2: Categories describing the different ways to understand the phenomenon *class* found in the group.

Many of the students express an understanding belonging to the first category. Student H says:

> A class is, well I figure a class is like a small program, that's how I think of it, a small program inside the whole big program, if you say that the big program is the main program, then the class is like a small program doing certain things.

The understanding has its focus on the program structure and the programmers task and describes the class-concept as a help for the programmer when structuring the code. It deals with the code and the programming task, and the description of the class reminds of a description of modules, even if no student explicitly uses this formulation. Some students emphasize this module aspect:

> C: Then the class should be something reasonable, containing what you detach [...] But the class I suppose, is only a diffuse collection of, what I belive belong together in some way.

The second category is the most common understanding expressed in the group. Even if none of the students explicitly uses the expression "abstract data type", the descriptions point in this direction.
Student O says:

> Eh, when you write a class [...], you write what you want the objects to look like, and that's how I understand a class, that you are able to create an object and something about what you want to do with this object in the different methods [...]

In the third category in Table 2, the close relationship between the class definition and the reality the class depicts is pronounced. This category includes the understanding expressed in category two. Only a few students express this kind of understanding.

> I: I mentioned class. How do you understand classes?
> C: It's a bit more diffuse actually. Class, it is I can imagine that a class contains, can contain a number of objects or only one object and different operations you can do in an object or between objects. So you

can also imagine what it would represent in the reality.
I: Okay.
C: Yes well, you can think of a workplace and a person working there, then you have two objects and then they can kind of interplay with each other through different operations sort of, what do I know. The person gets coffee and then the coffee variable decreases in the workplace like that.

4.3 Dimensions of Variation

The phenomenographic analysis of the data has revealed understandings found among the students. These are critical understandings from a student's perspective. Each aspect of the concept, expressed as categories of description in Table 1 and Table 2, require focal awareness of a specific dimension of relevance for the understanding.

There is a close relationship between the concepts *object* and *class*, and Table 1 and Table 2 show similar patterns for the understandings of the concepts. In the empirical data collected for the present study, most students express understandings of the concepts in corresponding categories. If a student for example expresses an understanding of *object* corresponding to the second category in Table 1, he or she also expresses an understanding of *class* corresponding to the second category in Table 2. There are few, if any examples where students show an advanced understanding of one concept, and a poor understanding of the other concept. The understandings found in the three categories in the two tables, will now be grouped together and discussed in terms of focal awareness found in the empirical data. The focal awareness of the understandings are then analysed in order to find dimensions of variations necessary for discernment of these aspects of the concepts. In this way we have identified the variation necessary for learning to take place.

In the first categories in Table 1 and Table 2 the students have experienced class as 'an entity of the program, contributing to the structure of the code', and object as 'a piece of code'. The focal awareness of this understanding of a class is the appearance of the structure of the program text. The focal awareness of the understanding of objects is on the program text. To be able to focus on this aspect, students need to discern that in different programs objects and classes appear in different ways. In that sense, the textual representation of programs constitutes a dimension of relevance for the understanding of *object* and *class*. Different, specific program texts constitute values along this dimension.

In the second category, class is experienced as 'a description of properties and behaviour of the object', where object is understood as 'something that is active in the program'. The focal awareness in these categories is on what happens during execution of the program, in particular on the objects created and how they contribute to different events at run-time. The objects are the active parts of the program, accomplishing the task given. To be able to discern the understanding expressed in the second categories, the students need to focus on the objects the program creates and events happening at execution of the program. Here, the relation between class description, object action, and resulting events during the execution of the program constitutes a dimension. Different specific cases of such relations provide values along this dimension. The variation between these values can enhance an awareness of object and class corresponding to the second category of understanding, according to Table 1 and Table 2.

In the last categories in Table 1 and Table 2, class is experienced as 'a description of properties and behaviour of the object', where object is understood as 'a model of some real world phenomenon'. The focal awareness is still on the class' description of the active objects, but now with an emphasis on the reality aspect of the class description. In this case, the relation between class, object and real-life phenomena constitute a dimension. Different specific cases of such relations constitute values along this dimension.

5. IMPLICATIONS FOR EDUCATION

Our results can shed new light upon and give explanation to other research and discussions in the field. The following paragraphs show some examples of this. Holland, Griffiths and Woodman list some misconceptions noticed at distance courses where Smalltalk was taught, in one introductory undergraduate course, and one postgraduate course [4]. One misconception mentioned is "object as a kind of variable". Students with previous experience of procedural programming may, if the examples they first come across have only one instance variable, develop the misconception that objects are in some sense mere wrappers for variables. It is trivially easy to avoid this misconception by ensuring that all the classes showed as an introduction, have more than one instance variable. Another misconception that can appear is if the data aspect of objects is overemphasized at the expense of the behavioural aspect. This misconception can be avoided by using introductory object examples where the response to a message is substantially altered depending on the state of the object. Both the misconception "object as a kind of variable" and the overemphasizing of the object's data aspect is an indication of the importance to attain a conception according to the second categories in Table 1 and Table 2. The second category in Table 1 emphasizes that objects are active during execution of the program. This points to the behavioral aspect of objects. The second category in Table 2 explains classes as a description of both data about the object, and methods explaining the behaviour of the object. As explained in section 4.3, the relation between class description, object action, and resulting events during program execution constitutes a dimension where variation is needed. This implies, e.g., variation in values of several instance variables, caused by several method calls. This is according to the recommendations from Holland et al.

A common problem among novice programmers, also mentioned by Holland et al, is to understand the difference between *class* and *object*. This is obviously a problem if several examples are presented in which only a single instance of each class is used. To avoid this, good practice is always to work with several instances of each class. As explained in section 4.3, the textual representation of programs constitutes a dimension of variation. This implies variation in the sense of presenting more than one instance of the class in the code, as recommended by Holland et al.

In the light of the present study, the recommendations from Holland et al can be summarized as: *variation in dimensions corresponding to critical aspects of the understanding is of great importance.* These dimensions of variation are not only pinpointed here, but also explained in the theory of phenomenography and in the analysis of the data by applying variation theory on the results of the study.

Holmboe [5] performed a study where a few people of different background were asked to describe in their own words what object-oriented programming is. He asked students who had just finished an introductory course on object-oriented programming, senior students tutoring the same course and professors of Computer Science or System Engineering. He made a qualitative analysis of the answers, and comments that some types of knowledge are more suitable as a basis for further knowledge construction than others. When analysing the results from the study he writes about understandings which include the world outside the computer itself: "A person with holistic knowledge relates the implementation and design of a computer program to the real world being simulated." Holmboe emphasizes the importance that "[...] more students will experience the connection between reality, model and implemented program, and thus reach holistic knowledge of object-orientation sooner in their learning process." The third categories in Table 1 and Table 2 capture an understanding of classes and objects that includes the world outside the computer itself. The dimensions of variation found and discussed in section 4.3 are valuable as a basis for teachers to facilitate for the students to reach this understanding.

In Fleury's study on students' constructed rules [3], she stresses, with a reference to Holland, Griffiths and Woodman [4], the importance of carefully constructed sample programs to avoid misconceptions of concepts. Our study stresses the importance of designing the education so that the students can discern the critical aspects of the understanding. Carefully constructed sample programs in this sense means variation of dimensions corresponding to these critical aspects. This is applicable not only on sample programs, but in all different aspects of the learning environment.

6. CONCLUSIONS

For the Java educator, one challenge is to construct an educational environment which facilitates for students to reach a rich understanding of the concepts *object* and *class*. To this end it is important to know the different ways in which students (as opposed to experts) typically experience these concepts. Our phenomenographic study has given such insight. Next, the educator needs to identify what variation the students have to discern in order to become aware of aspects belonging to a rich understanding of these concepts. Here, variation theory can be a useful tool, as demonstrated in the previous discussion.

By using dimensions of variation, discussed in the previous section, implications for teaching are found. Teaching is here defined in a wide sense. By teaching we mean everything that supplies resources for learning. Examples of such resources could be programming assignments, software tools, lectures, Internet and fellow students, anything the students choose to use in their learning. The whole organisation of the learning environment is in this sense teaching.

A general implication for teaching is to make resources in the learning environment available that help students to discern the aspects mentioned Table 1 and Table 2 and developed in the previous section. These are resources that point out the corresponding dimensions of variation of the aspects.

The results in Table 1 and Table 2 can be implemented in the teaching and learning environment offered to the students, in a number of ways. There is a great freedom and possibility to adapt the results to the preferences of each teacher and student group.

7. REFERENCES

[1] S. A. Booth. *Learning to Program. A phenomenographic perspective*. Number 89 in Göteborg Studies in Educational Science. Acta Universitatis Gothoburgensis, Göteborg, Sweden, 1992.

[2] A. E. Fleury. Student conceptions of object-oriented programming in java. *The Journal of Computing in Small Colleges*, 15(1), November 1999.

[3] A. E. Fleury. Programming in java: Student-constructed rules. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 2000.

[4] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. In *ACM SIGCSE Bulletin , Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education, Volume 29 Issue 1*, 1997.

[5] C. A. Holmboe. Cognitive framework for knowledge in informatics: The case of object-orientation. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, 1999.

[6] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOURNAL OF OBJECT-ORIENTED PROGRAMMING*, January 1999.

[7] S. Kvale. *InterViews: An introduction to qualitative research interviewing*. Sage, 1996.

[8] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Ass., Mahwah, NJ, 1997.

[9] B. Meyer. *Object-oriented Software Construction*. International series in Computer Science. Prentice Hall, 1988.

[10] M. F. Pang. Two faces of variation: on continuity in the phenomenographic movement [1]. *Scandinavian Journal of Educational Research*, 47(2), 2003.

Paper II

Putting Threshold Concepts into Context in Computer Science Education

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269 USA
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Mark Ratcliffe
Department of Computer Science
University of Wales
Aberystwyth, Wales
mbr@aber.ac.uk

Kate Sanders
Department of Math and Computer Science
Rhode Island College
Providence, RI USA
KSanders@ric.edu

Carol Zander
Computing & Software Systems
Univ. of Washington, Bothell
Bothell, WA, USA
zander@u.washington.edu

ABSTRACT

This paper describes Threshold Concepts, a theory of learning that distinguishes core concepts whose characteristics can make them troublesome in learning. With an eye to applying this theory in computer science, we consider this notion in the context of related topics in computer science education.

Categories and Subject Descriptors

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

General Terms

Theory

Keywords

Threshold Concepts, Education research, Constructivism

1. INTRODUCTION

As educators, we are aware of topics that are difficult for our students to learn, yet necessary for their development as computer scientists. Meyer and Land [25, 26] have proposed using what they call "Threshold Concepts" as a way of characterizing particular concepts that might be used to organize and focus the educational process. These are concepts whose achievement is necessary to making progress in the discipline, that transform the way a student looks at a discipline, but are also places in the curriculum where students get stuck, unable to make progress until they become unstuck.

In this paper, we look at the defining characteristics of Threshold Concepts in general, then place them in the context of related work in the computing discipline.

1.1 Characteristics of Threshold Concepts

Threshold Concepts are a subset of the core concepts in a discipline. Core concepts are building blocks that must be understood; Threshold Concepts, in addition, are [25]:

1. *transformative*: they change the way a student looks at things in the discipline.
2. *integrative*: they tie together concepts in ways that were previously unknown to the student.
3. *irreversible*: they are difficult for the student to unlearn.
4. potentially *troublesome* for students: they are conceptually difficult, alien, and/or counter-intuitive.
5. often *boundary markers*: they indicate the limits of a conceptual area or the discipline itself. Students who have mastered these Threshold Concepts have, at least in part, crossed over from being outsiders to belonging to the field they are studying.[1]

These criteria are all closely related: if something truly transforms the way you look at your discipline you are unlikely to forget it; if something integrates concepts in previously unknown ways it transforms the way you see those

[1]In listing the criteria for Threshold Concepts, some have used the term "bounded" for the concepts themselves.It seems clear, however, that what is meant is that the discipline, or part of it, is bounded *by* the Threshold Concept, and the concepts are boundary *markers*. For example, Davies states, "[A] threshold concept is bounded. That is, it helps to define the boundaries of a subject area." [8, p. 5].

ITiCSE'06, June 26–28, 2006, Bologna, Italy.

topics; and the process of understanding something that is alien or counter-intuitive may well require a mental transformation.

These concepts, then, are thresholds that our students must cross, but where many of them get stuck. They are both conceptually difficult and lead to a broader (and necessary) understanding of the discipline. If we can identify these concepts, they can provide focus points within the curriculum: places where we might place extra effort to help students avoid getting stuck, and thus continue to make progress.

Particular Threshold Concepts have been suggested in a number of disciplines: *limit* in Mathematics, *opportunity cost* in Economics, *irony* in Literary criticism, *depreciation* in Accounting, e.g. Identifying such concepts in different disciplines may indicate whether this is a generally useful approach or one that applies only to fields with certain characteristics. We are interested, naturally, in computer science.

While the notion of Threshold Concepts is new to computer science[23], it relates to a number of other CS education research areas. In this paper, we consider some related areas of research, and show how these relate to (and differ from) Threshold Concepts. In particular, we discuss the following:

- constructivism, particularly in CS education;
- mental models;
- student misconceptions;
- breadth-first approaches to introductory computer science, an approach built around important computing concepts;
- Fundamental Ideas, an alternative organizational framework based on concepts that develop and persist through the curriculum; and
- two specific concepts, abstraction and object-orientation; specifically why they are good candidates for computer science Threshold Concepts.

2. CONSTRUCTIVISM

Constructivists interpret student learning as the development of personalised knowledge frameworks that are continually refined. According to this theory, to learn, a student must actively construct knowledge, rather than simply absorbing it from textbooks and lectures [9]. Students develop their own self-constructed rules, or "alternative frameworks"[1]. These alternative frameworks "naturally occur as part of the transfer and linking process"[7]. They represent the prior knowledge essential to the construction of new knowledge[32]. When learning, the student modifies or expands his or her framework in order to incorporate new knowledge.

Constructivism is a theory of learning; Threshold Concepts are points where students have difficulty learning. In constructivist terms, Threshold Concepts are distinctive parts of a student's knowledge framework, parts that make a computer scientist's knowledge framework different from that of someone who has not studied computer science. They connect other parts of the framework together, and they are parts of the framework that are particularly troublesome, or difficult to build. The difference between students' and instructors' knowledge frameworks, like Threshold Concepts, help to explain the "knowledge barrier" that exists between students and instructors. Threshold Concepts are core learning outcomes about which there is some general agreement in the discipline, so they may be more objective than knowledge frameworks; on the other hand, it may well be that some students experience a given concept as a threshold, and others do not.

Threshold Concepts, as developed by Meyer and Land, are closely tied to the constructivist tradition. Indeed, their use of "troublesome" follows from Perkins [28] discussing challenges that constructivists must face.

3. MENTAL MODELS

A theory of how our brains work when we reason about the world, well known in cognitive psychology, is that we form mental models [19]. Briefly, a mental model is one person's internal model of a system's properties and behavior. The use of a mental model makes it possible to predict the system's response to various actions and thus makes it possible for an individual to select the best possible action [20].

The consequence of this is that a "faulty" mental model can lead to "faulty" actions. A classic example: to increase the temperature in a room as fast as possible a common action is to turn the room thermostat up as much as possible, based on the expectation that a thermostat works the same way as a water tap. However, since a thermostat works like a switch, the temperature increases at the same rate as long as the setting is above the temperature of the room. Similarly, a faulty model of how various programming constructs work will cause problems when developing software.

Ben-Ari [1] argues that the lack of mental models plays an important part in why students find it difficult to learn how to program. His argument is that, having no previous models to build on, programmers are forced to construct their own mental models from scratch. Wiedenbeck [34] investigated how novice programmers' mental models of their programs depend on whether a procedural or an object-oriented language was used. Similarly, Yehezkel et al [35] describe the importance of forming a mental model of a system in order to understand it. Wiedenbeck et al [33] claim that the ability to form mental models is a predictor for course outcome.

In these terms, Threshold Concepts are points at which students may have trouble forming their mental models. Mastering the relevant threshold concepts may make it possible to form a mental model, or change one mental model into a new one. For example, a possible Threshold Concept could be the difference between "pass-by-value" and "pass-by-reference", understanding this difference will have a profound impact on how a person would understand the semantics of a programming language and model the data flow in a program.

While there are similarities between threshold concepts and mental models – both can for example be transformative – there are also major differences. Threshold Concepts are troublesome to learn while some mental models can be learned without much effort, for example modeling the computer screen as a desktop. Threshold Concepts are accepted concepts within a discipline, while mental models are subjective and individual.

4. STUDENT MISCONCEPTIONS

Research into misconceptions, like Threshold Concepts, focuses on the ways in which students *fail* to learn. Over the past almost 30 years, this work has had considerable impact [32]. According to constructivist theory, misconceptions naturally occur as students modify and extend their knowledge frameworks to learn new topics. For example, an individual's previous understanding can lead to misconceptions when familiar terms are used in unfamiliar contexts. Bonar and Soloway [3] use the *while* statement to demonstrate the problem of linguistic transfer. In common language the *while* can imply continual testing of the condition (*e.g.*, "hold your breath while underwater"). In programming loops the time of the test is limited: it occurs once only on each iteration. Students who interpret the test as continual have a misconception. The overloading of language, mathematical symbols and previous programming experience, taking information from one context and using it in another have all been demonstrated to cause misconceptions [7].

Misconceptions can be integrative, like Threshold Concepts (although the integration is not necessarily correct), but can also result from failing to integrate knowledge. As Eylon and Lynn [14] have observed, students deal with apparent contradictions by keeping their knowledge isolated. This might explain why students often fail to transfer knowledge from one course to another.

Misconceptions, like Threshold Concepts, can be hard to forget. As students familiarize themselves with new topics, their partial knowledge leads them to develop their own rules [15]. Unfortunately because the knowledge is incomplete these self-constructed rules may result in misconceptions, which once established are difficult to change. These premature generalizations are then used to filter and distort new information often compounding the misconception [18]. In order to realign such robust ideas, significant effort is required, undertaking radical reordering of the concept [24].

Unlike Threshold Concepts, misconceptions are certainly not core concepts that we want our students to learn. Moreover, we can speculate that they are not troublesome to learn. Especially if the correct (threshold) concept is difficult or counter-intuitive, the misconception may be much easier to learn.

5. BREADTH-FIRST INTRODUCTION

Threshold Concepts are a subset of the core concepts in the discipline. Thus, a list of the core concepts would be a good starting point in identifying the Threshold Concepts.

One source for such a list is the breadth-first approach to teaching computer science, since such courses are often based on important ideas or topics in computer science The concepts covered vary from course to course, but can include decision trees, number representation, patterns in programming, divide-and-conquer, recursion, the Church-Turing thesis, the von Neumann architecture, time complexity, intractability, types and values, classes and objects, design, encapsulation, inheritance, polymorphism, program correctness, iteration, recursion, conceptual and formal models, levels of abstraction, reuse, and tradeoffs.[2, 5, 10, 30].

The only criteria for these concepts is that they be, in terms of the definition of Threshold Concepts, core concepts in computer science. While most computer scientists would agree on the general importance of these concepts, they are probably not all Threshold Concepts. A concept such as "divide and conquer," for example, integrates many areas of computer science, but it does not appear to be troublesome for students to understand. Similarly, the idea of reuse is an easy one to grasp (even though it may be harder to implement consistently). The concepts on the breadth-first courses' lists may not be transformative, and some of them are probably all too easy to forget. Which, if any, of them qualify as Threshold Concepts is a question for empirical investigation.

6. FUNDAMENTAL IDEAS

Schwill[31] has proposed organizing the computing curriculum around another set of core concepts, the Fundamental Ideas, a set of ideas that are central to the discipline. This follows from the work of Bruner [6], who proposed that science teaching should be organized around the structure of science, as expressed by its fundamental ideas. These are ideas with broad applicability, and that can be taught at multiple levels within the curriculum, from early to advanced, at increasingly sophisticated levels.

Drawing on work applying Fundamental Ideas to mathematics, Schwill proposes four criteria for these ideas in computer science (paraphrasing):

> **Horizontal criterion** The idea is applicable or observable in multiple ways and multiple areas of CS; it organizes and integrates multiple phenomena.
>
> **Vertical criterion** The idea can be taught at every intellectual level, at different levels of sophistication.
>
> **Criterion of time** The idea is clearly observable in the history of computer science, and will be relevant for a long time.
>
> **Criterion of sense** The idea also has meaning in everyday life, and can be described in ordinary language.

These can be used to organize and relate subjects in computer science. Fundamental Ideas are taught throughout the curriculum, and when new concepts are presented, they are related to the appropriate Fundamental Ideas that the students know, thus providing context. Moreover, relating new concepts to these ideas should further develop the ideas, so the learning process can be seen as gradually gaining a greater understanding of these Fundamental Ideas.

There is some obvious overlap between Fundamental Ideas and Threshold Concepts. Both are integrative, and both include topics that should be understood by any competent computing professional. But the Fundamental Ideas are likely *not* transformative, in that they are gradually developed from common-sense understanding of everyday phenomena. Threshold Concepts, on the other hand, may not be teachable at every level. Finally, Fundamental Ideas are clearly not boundary markers, given the *Criterion of sense*, as these ideas have everyday out-of-discipline meanings.

Overall, these approaches seem to be orthogonal. Threshold Concepts are based on transformative events, while Fundamental Ideas are based on long-term development. It seems likely that any given Threshold Concept could be described in terms of the related Fundamental Ideas, and that there are Threshold Concepts that appear at points in the

development of a given Fundamental Idea. Threshold Concepts identify the discontinuities in a student's development, while Fundamental Ideas identify different ongoing threads in this process which may or not have such discontinuities.

7. PARTICULAR CONCEPTS

For illustration, we consider two particular candidate Threshold Concepts, abstraction and object-orientation, in relation to the criteria for Threshold Concepts.

7.1 Abstraction

The ability to abstract, and more than that, to move flexibly from one level of abstraction to another, is a key skill in computer science. As noted in Section 5, abstraction is a core concept, and and it has been widely studied. If one searches the papers available through the ACM Digital Library (which includes over 200 computing journals) using the keyword "abstraction", 63% of all articles are found.

Or-Bach and Lavy [27] construct a cognitive task analysis taxonomy regarding abstraction and inheritance. They find that abstraction is key with relation to object-oriented programming and determine that it is a higher order cognitive skill difficult for students to conceptualize.

Similar findings about students' use of methods and attributes with regard to abstraction were found also by Detienne [11] who claims that one of the main difficulties experienced by novices is the articulation between declarative and procedural aspects of the solution. While an object can be thought of as an abstract data type, in object-oriented design it seems appropriate to consider the abstraction inherent in object orientation as behavior abstraction. This understanding is seen in advanced OO designers, but not in novices.

Box and Whitelaw [4] argue that abstraction helps explain the relative difficulty of learning object-oriented technology. In this technology, the learning of new concepts include several steps, and abstraction is both the last and most difficult step. Significant parts of this abstraction are the decisions as to which entities are to be grouped together and which attributes are to be ignored or parameterized.

Hadjerrouit [17] writes about students' learning of the object-oriented programming language Java:

> ...to understand Java concepts properly, problem solving should begin at the conceptual level, not at the code level where programming becomes the main issue. Furthermore, substantial attention should be devoted to the meta-level process required to develop solutions.

Hadjerrouit expresses a constructivist perspective, and a clear emphasis on the importance of gaining an understanding of the abstract concepts as well as the concrete.

7.2 Object-orientation

Much has been reported on experiences with teaching the object-oriented paradigm. It is widely acknowledged that Object-oriented programming is difficult to teach [21].

Here we consider Object-orientation at its most basic: including *objects, classes,* and *encapsulation,* but ignoring such things as inheritance and polymorphism. Even at this simplified level, it is a core concept in computer science, the first one learned by many introductory students, and necessary for students to understand. Holland, Griffiths and Woodman [18] claim that misconceptions of object concepts can be hard to shift later. Such misconceptions can act as barriers through which later all teaching on the subject may be inadvertently filtered and distorted.

The literature suggests that this satisfies the requirements for a Threshold Concept. We consider these requirements in turn.

There is much evidence in the literature that students find basic object-orientation *troublesome* to learn. Eckerdal and Thuné [13] interviewed first year students who had just finished their first programming course on their understanding of the concepts object and class. Many students stated that they found the concepts troublesome to learn despite great effort to understand them. Ragonis and Ben-Ari [29] studied high-school students in a first programming course in Java. They found that one of the major difficulties is to understand the creation of object by constructor. Fleury [16] has investigated Java students comprehension of encapsulation and reuse of code. One result she reports is that many students consider reducing the number of lines of code and reducing the number of classes to be more important than encapsulation. Many students are annoyed by the "jumping around" necessary when reading programs with multiple classes.

There is also evidence that object-orientation is *transformative.* Luker [22] argues that learning the object-oriented paradigm, "requires nothing less than complete change of the world view". Eckerdal [12] reports that some students, who had used other programming paradigms before, could use the concepts object and class in a way that made programming more efficient.

Luker [22] furthermore discusses how encapsulation ties together the concepts object and class. Eckerdal [12] found that students had problems separating the concepts object and class. These suggest that basic object-orientation *integrates* these concepts, and gives some justification for our considering them together.

8. CONCLUSIONS

In this paper, we discussed the idea of Threshold Concepts as a possible way to organize and focus learning in computer science. Moreover, we tried to put it in context with other areas of computer science education. Summarizing,

> **Constructivism** seems to be assumed by Threshold Concepts: learning these concepts are particularly interesting places in the process.
>
> **Mental Models:** Threshold Concepts are places where fundamentally different mental models are developed, often with difficulty.
>
> **Misconceptions** The transformational nature (and difficulty) of attaining Threshold Concepts make them obvious places where misconceptions can be formed; observed misconceptions suggest places to look for Threshold Concepts.
>
> **Breadth-first introductions** involve teaching a broad range of "important" core concepts; as Threshold Concepts are also a subset of the core ideas there may be overlap.
>
> **Fundamental ideas** seem to be an orthogonal organization principle, emphasising the long-term develop-

ment of central ideas rather than just the transformational points.

Abstraction and Object-orientation are two possible concepts that may be Threshold Concepts; certainly evidence exists that they have the appropriate characteristics.

Threshold Concepts may prove to be useful in CS Education once we identify them. Previous work in Computer Science Education suggests places that we might reasonably start to look.

9. REFERENCES

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

[2] A. Biermann. *Great Ideas in Computer Science: a gentle introduction*. MIT Press, 1990.

[3] J. Bonar and E. Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway and J. Spohrer, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, 1989.

[4] R. Box and M. Whitelaw. Experiences when migrating from structured analysis to object-oriented modelling. In *Proceedings of the Australasian conference on Computing education*, pages 12–18. ACM, 2000.

[5] J. G. Brookshear. *Computer Science: an overview*. Addison Wesley, sixth edition, 2000.

[6] J. Bruner. *The process of education*. Harvard University Press, Cambridge, MA, 1960.

[7] M. Clancy. Misconceptions and attitudes that interfere with learning to program. In S. Fincher and M. Petre, editors, *Computer Science Education Research*. Taylor and Francis Group, London, 2004.

[8] P. Davies. Threshold concepts: how can we recognise them. 2003. Paper presented at EARLI conference, Padova. Downloaded from http://www.staffs.ac.uk/-schools/business/iepr/docs/etcworkingpaper(1).doc.

[9] R. Davis, C. Maher, and N. Noddings. Constructivist views of the teaching and learning of mathematics. *J. Res. Math. Teaching, Monograph No.4*, 1990.

[10] P. Denning. Great principles in computing curricula. *SIGCSE Bull.*, 36:336–341, 2004.

[11] F. Detienne. Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*, 9:47–72, 1997.

[12] A. Eckerdal. On the understanding of object and class. Technical Report 2004-058, Dept. of Information Technology, Uppsala Univ., 2004.

[13] A. Eckerdal and M. Thuné. Novice java programmers' conceptions of "object" and "class", and variation theory. In *Proc. ITiCSE '05*, pages 89–93, 2005.

[14] B. Eylon and M. Linn. Learning and instruction: An examination of four research perspectives in science education. *Rev. Educational Research*, 58(4), 1988.

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

[16] A. E. Fleury. Encapsulation and reuse as viewed by Java students. *SIGCSE Bull.*, 33(1):189–193, 2001.

[17] S. Hadjerrouit. A constructivist framework for integrating the java paradigm into the undergraduate curriculum. *SIGCSE Bull.*, 30(3):105–107, 1998.

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

[19] P. N. Johnson-Laird. *Mental models: towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983.

[20] D. E. Kieras and S. Bovair. The role of a mental model in learning to operate a device. *Cognitive Science*, 8:255–273, 1984.

[21] M. Kölling. The problem of teaching object-oriented programming, part I: Languages. *J. Object-oriented Programming*, 11(8):8–15, 1999.

[22] P. A. Luker. There's more to OOP than syntax. *SIGCSE Bull.*, 26(1):56–60, 1994.

[23] R. McCartney and K. Sanders. What are the "threshold concepts" in computer science? In *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, page 185, 2005.

[24] M. McCracken, W. Newstetter, and J. Chastine. Misconceptions of designing: a descriptive study. *SIGCSE Bull.*, 31(3):48–51, 1999.

[25] J. Meyer and R. Land. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. ETL Project Occasional Report 4, 2003. http://www.ed.ac.uk/etl/docs/ETLreport4.pdf.

[26] J. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49:373–388, 2005.

[27] R. Or-Bach and I. Lavy. Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bull.*, 36(2):82–86, 2004.

[28] D. Perkins. The many faces of constructivism. *Educational Leadership*, 57(3):6–11, 1999.

[29] N. Ragonis and M. Ben-Ari. Teaching constructors: A difficult multiple choice. In *16th European Conference on Object-Oriented Programming, Workshop 3*, 2002.

[30] G. M. Schneider and J. L. Gersting. *An Invitation to Computer Science*. Brooks Cole, second edition, 1998.

[31] A. Schwill. Fundamental ideas of computer science. *Bull. European Assoc. for Theoretical Computer Science*, 53:274–295, 1994.

[32] J. Smith, A. diSessa, and J. Roschelle. Misconceptions reconceived: A constructivist analysis of knowledge in transition. *J. Learning Sciences*, 3(2), 1993.

[33] S. Wiedenbeck, D. LaBelle, and V. N. R. Kain. Factors affecting course outcomes in introductory programming. In *16th Annual Workshop of the Psychology of Programming Interest Group*, 2004.

[34] S. Wiedenbeck and V. Ramalingam. Novice comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Human-Computer Studies*, 51:71–87, 1999.

[35] C. Yehezkel, M. Ben-Ari, and T. Dreyfus. Computer architecture and mental models. *SIGCSE Bull.*, 37(1):101–105, 2005.

Paper III

Threshold Concepts in Computer Science: Do they exist and are they useful?

Jonas Boustedt
Department of Mathematics, Natural, and Computer Science Högskolan i Gävle
S80176 Gävle, Sweden
bjt@hig.se

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT USA
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Mark Ratcliffe
Department of Computer Science
University of Wales
Aberystwyth, Wales
mbr@aber.ac.uk

Kate Sanders
Department of Math and Computer Science
Rhode Island College
Providence, RI USA
ksanders@ric.edu

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA USA
zander@u.washington.edu

ABSTRACT

Yes, and Yes.

We are currently undertaking an empirical investigation of "Threshold Concepts" in Computer Science, with input from both instructors and students. We have found good empirical evidence that at least two concepts—Object-oriented programming and pointers—are Threshold Concepts, and that there are potentially many more others.

In this paper, we present results gathered using various experimental techniques, and discuss how Threshold Concepts can affect the learning process.

Categories and Subject Descriptors

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

General Terms

Measurement, Experimentation

Keywords

Threshold Concepts, learning theory

SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.

1. INTRODUCTION

Computer science is a young, rapidly changing discipline; we have had relatively few years to study the ways in which students learn and how to help them most effectively. Meyer and Land [13] have proposed using "Threshold Concepts" as a way of characterizing particular concepts that might be used to organize the educational process. The idea has the potential to help us focus on those concepts that are most likely to block students' learning [4]. A subset of the core concepts in a discipline, Threshold Concepts are defined by Meyer and Land [13] are:

- *transformative*: they change the way a student looks at things in the discipline.
- *integrative*: they tie together concepts in ways that were previously unknown to the student.
- *irreversible*: they are difficult for the student to unlearn.
- potentially *troublesome* (as in [15]) for students: they are conceptually difficult, alien, and/or counter-intuitive.
- often *boundary markers*: they indicate the limits of a conceptual area or the discipline itself.

This paper describes an ongoing project aimed at empirically identifying Threshold Concepts in computer science. In a multi-national, multi-institutional study, we have gathered data from both educators and students. The paper outlines experimental techniques, issues raised, and results to date. Information about how Threshold Concepts fit with other learning theories and how they can be put into context

with other ways to organize computer science concepts can be found in [5].

Section 2 describes the techniques we have used to gather data about potential Threshold Concepts from the perspective of both instructors and students. In Section 3 we present the results of a preliminary analysis of the data, including the identification of two Threshold Concepts in computing with some associated evidence. Section 4 looks at implications for educators and finally, in Section 5, we present our preliminary conclusions and discuss the future directions of this research effort.

2. DATA GATHERING AND ANALYSIS

The study began by gathering data from instructors in computer science. This was followed by a brief analysis and a much more in-depth study of graduating seniors.

2.1 The instructors: informal interviews and surveys

In June 2005, at the Conference on Innovation and Technology in Computer Science Education (ITiCSE), 36 instructors from nine countries were interviewed and asked for suggestions of concepts that meet the criteria for a Threshold Concept. These interviews were unstructured and done in a fairly conversational style.

From these, we gained much insight. First, the idea of Threshold Concepts is compelling: nearly everyone we spoke with was immediately interested. In total 33 concepts were suggested, with the most popular being: levels of abstraction; pointers; the distinction between classes, objects, and instances; recursion and induction; procedural abstraction; and polymorphism. Second, while some concepts came up again and again, there was no universal consensus.

In November 2005, we gave a poster and had discussions with researchers at the Koli Calling 2005 Conference on Computer Science Education in Finland [12] and used a questionnaire and interviews to gather data more systematically from conference participants. The results were similar to those collected at ITiCSE but it was quite apparent that instructors focus on "difficult to learn" more than any other aspects of the concepts they discuss.

2.2 The students: semi-structured interviews

Given the tentative list of concepts derived from instructors and the literature ([1, 2, 3, 17, 18] e.g.), we began to investigate the question of whether these—or any—concepts are experienced by students as thresholds.

We chose to initially interview graduating students, since they were more likely than novices to have mastered the relevant concepts, and to have developed some perspective. So far we have completed 16 interviews with students at seven institutions in a total of three countries. The script for these interviews is given in Figure 1.

We started by addressing the *troublesome* criterion, asking students for concepts they found difficult at first (places where they were initially "stuck"). From these, we selected one concept to pursue in depth and addressed the *transformative, integrative,* and *irreversible* criteria in that context. We did not examine the *boundary marker* criterion, as it is related more to disciplinary boundaries and less to individual experience. The interviewers agreed in advance on a list of five Threshold Concept candidates (control structures, thinking sequentially, parameters, objects, memory model)

1. Could you tell me about something where you were stuck at first but then became clearer? *(Subject answers <X>.)*

   The rest of this session will now focus on <X>.
2. Can I start by asking you to tell me your understanding of <X>?
3. Assume that you were explaining <X> to someone just learning this material, how would you do it?
4. Tell me your thoughts, your reactions, before, during and after the process of dealing with <X>.
5. Can you tell me what helped you understand <X>?
6. Can you describe how you perceived/experienced <X> while you were stuck and how you perceived/experienced it afterwards?
7. Based on your experience, what advice would you give to help other students who might be struggling with <X>?
8. Please tell me what other things you need to understand in order to gain a good understanding of <X>.
9. Can you tell me how your understanding of <X> has affected your understanding of other things?
10. Was your understanding of <X> something that you had to keep reviewing or having learned it once were you OK with it?
11. Describe how and in what context you have used <X> since you learned it?
12. Is there something more you want to tell me about <X>?
13. To finish the interview, can you tell me whether there are any other things where you were stuck at first but then became clearer? I promise I won't ask you about them in detail!

Figure 1: Script Excerpt for Student Interview

that had come up repeatedly in instructor interviews. If the student mentioned one of those, that concept was chosen.

The aim of this deeper investigation was twofold. It enabled us to gather evidence as to whether specific concepts met the requirements for Threshold Concepts (Questions 1, 4, 6, 8, 9, 10 in Figure 1). In addition, it gave us data for an analysis (in progress) of graduating students' understanding of central concepts (Questions 2, 3, 5, 12 in Figure 1).

For analysis, the student interviews were transcribed verbatim and where necessary, translated into English by the interviewer.

3. THRESHOLD CONCEPTS IN COMPUTER SCIENCE

Of the concepts discussed in depth by the students, we selected two that seemed promising based on the interview content and closely examined the interviews regarding those concepts. For both concepts – object-oriented programming (OOP) and pointers – we found evidence that they satisfy the criteria for Threshold Concepts.

3.1 Object-oriented programming

Object-oriented programming is experienced as difficult both to teach [8, 16] and to learn [9, 19]. When Eckerdal and Thuné [6] interviewed first-year students after their first programming course on their understanding of object-oriented concepts, many students stated that they found the concepts troublesome despite great effort to learn them.

The interviews give further evidence of OOP as a Threshold Concept. One student subject discussed OOP as *troublesome* to learn:

> *Subject8:* Stuck at first – I would have to say the initial object-oriented programming. Knowing how classes communicate, how you communicate between classes and really understanding how objects work...

The researcher performing the interview asks the subject about the *integrativeness* of OOP later in the interview by asking if once they understood OOP, were there other things they then understood. The subject discusses a multi-threaded programming course.

> *Subject8:* Well, for instance, the class we did for software engineering, what we did right now, the server that I wrote, each client that connects to the server, I thought of it almost as an object, which it is basically. And then that client connection would be held on to while waiting for other connections. And then there'd be this huge array of connections. And then, I mean, that wasn't that difficult for me to grasp that concept just because we'd kind of went over it in class, but I just think understanding object-oriented programming helped me to understand that there was this group of objects, group of threads, group of clients, whatever.
>
> *Interviewer:* That they were all working together?
>
> *Subject8:* That were all working together, exactly. And understanding object-oriented programming, I think, made that easy to learn; easy to understand.

This subject learned about OOP, then later learned about multi-threaded programming, and perceived a real connection at a fairly abstract level.

Another student also indicated that object-oriented programming is *troublesome* to learn:

> *Subject6:* [...] object oriented programming was one thing for example that took a long time before ... it clicked. Why and how it should be used.

The researcher later seeks to find out from the same student if object-oriented programming is *irreversible* by asking if the student had to review the OOP material.

> *Subject6:* Yes, I need to review sometimes, this is completely clear [...], often it's just syntactical details, [...] basic stuff is there, I've mainly used Java so, sure I'm a little bit stuck in those tracks but I can usually bring everything with me and just transfer it to C++ for example, [...]
>
> *Interviewer:* The big stuff is there so to speak
>
> *Subject6:* Yes, I can get, sometimes it can take, or yes some mistakes before I remember that, "right, those", [...] I can forget, to make some mistakes in the beginning but as long as I've once known and done it correctly some time then it usually comes back.

The student explains that some syntactic details might be forgotten in a specific object-oriented programming language, but "the basic stuff is there" (the object oriented paradigm) and "it usually comes back", implying object-oriented programming is *irreversible*.

The quote also shows the close relation between the *irreversible* and the *integrative* aspects of this Threshold Concept. The student explains that he/she can use the knowledge gained from programming in one language and transfer it to another language.

Later in the interview the researcher discusses the *transformative* aspect of learning object-oriented programming by asking about the difference in how the student looks at problems and their solutions before and after:

> *Subject6:* Yes, it's like day and night, before I came here I had ... I couldn't ... abstract the problems on, well to a very small extent perhaps, but today it's ... I can identify the problems usually in a very short time, unless it's very complex and difficult to understand but today I only see small sub-problems and ... usually simple solutions to them. Before it was just one large program that ... I solved sequentially in some way and the programs looked like that ...

Discussing the same topic the interviewer later asks about problem solving, about what role OOP played:

> *Subject6:* It simplifies it, even if I don't use an OO language the OOP way of thinking can help a lot in ... in some way ... well, you can give a lot of data, you can give it some kind object status even if it doesn't have its own methods etc, in that way...

The student explains that he/she looks at programming in a completely new way after learning object-oriented programming. The knowledge has *transformed* how the student looks at problems. This is consistent with Luker [10], who argues that learning the object-oriented paradigm, "requires nothing less than [a] complete change of the world view."

3.2 Pointers

The second concept identified as a potential Threshold Concept was the use of pointers, particularly when used as parameters. That this concept can be *troublesome* is illustrated by the following excerpt:

> *Subject13:* And so when you implement pointers and see then you're like okay I need to figure out how I modify that and it affects the memory. And then if I reference the memory I get what back. And then you start passing the arguments. And you have to understand passing by reference or passing by value. And a lot of those were definitely big hurdles right in the beginning because I didn't – it was just – I guess too theoretical of a concept for me to really put in practical sense.

The student describes the difficulties in understanding pointers in general and how they work with parameter passing.

Another student affirms that pointers can be *troublesome*:

> *Subject3:* I know that pointers are something that a lot of students have trouble with. [...]

Not only can we see that this student found pointers *troublesome*, but also that they are *integrative* and *transformative*:

> *Subject3:* And I think once you've realized that a pointer is just pointing to a place in memory, it's just pointing to a location, that's all it is. Then I think everything will flow from that. Yeah, because you realize then that the object itself is just a place in memory too.
>
> *Interviewer:* So before you weren't even thinking about memory so much.
>
> *Subject3:* Exactly. I didn't at all. Like in Java, I didn't think about memory nearly so much. I mean I knew that certainly memory was allocated, that memory was allocated with its variables and attributes and that kind of stuff. But I didn't ... when I was writing a program, I didn't ever think about what was happening underneath. Especially garbage collection.

Once the student understood how pointers work he/she was able to use this knowledge to explain how objects and references are implemented, thus getting an improved understanding of how Java works.

In another part of the interview the same student described how the understanding of pointers has helped in other subjects:

> *Subject3:* Well, as I was saying, in the hardware class and in Operating Systems, we definitely discussed pointers and I used it both conceptually and also in, well not in in Operating Systems, but in the hardware class in assembly language, we definitely used pointers.
>
> We definitely were dereferencing all the time in assembly language, so when we were, for example, writing to an address register, we would have to dereference it in order to get at the address to find out what was going on at that particular address in memory. So, definitely I used it again and again.

The clearest statement that the concept of pointers is *unforgettable* comes from an interview with Subject13 when asked if the understanding gained needed to be reviewed to be remembered:

> *Subject13:* The syntax I would have to review, guaranteed. The syntax is –it's a little – it's syntax. But the basic idea of passing by reference or value; no, once I understood that I – every time it's mentioned I immediately know and understand – I can see a picture – a diagram in my head of what I'm supposed to do. What – how the effect will work. So the concept was not lost at all.

4. IMPLICATIONS FOR EDUCATORS

By their nature, Threshold Concepts have a number of implications for educators. These are key concepts that students must understand to make progress in computer science: failing to gain this understanding and the associated lack of progress can lead to frustration, a poor understanding on how the discipline fits together, and increased attrition of students. It is important, therefore, that we understand these concepts, and how they are learned, in detail. It is useful to know that students tend to become stuck on a particular concept, but the deeper understanding of the student experience—how students get unstuck, and why some students get unstuck (or perhaps never get stuck at all) while others remain stuck—should provide ideas on how to help students to make progress toward understanding that concept. Knowing what concepts a particular Threshold Concept integrates can provide the instructor with a context in which the concept might effectively be taught.

Davies suggests that when teachers proceed on the incorrect assumption that students have learned a Threshold Concept, it may cause students to go forward with surface-level learning:

> In the absence of this understanding students can only resort to learning surface routines and language in the hope that they can pass this off as real understanding.[4]

More interesting still, a result found in economics ([14], as reported in [13]) was that teaching Threshold Concepts in simplified ways can have similar bad effects: the simplified version may be treated as "ritualized" knowledge–a superficial understanding–that makes it *more* difficult for the student to ultimately learn the concept. What these results suggest is how important it is that teachers accurately monitor the level of understanding of these concepts by their students, expecially if "going through the threshold" (in a deep-learning sense) is necessary for their students to progress.

Looking on the bright side, however, Threshold Concepts can provide positive opportunities to instructors. First, they may help us manage the ever-growing curriculum. For example, Computer Curricula 2001 (CC2001) [7] included 63 core units, each made up of several topics. If we can identify a relatively small number of Threshold Concepts within the curriculum, instructors will be able to focus their efforts on helping students with those concepts. Second, because Threshold Concepts are integrative, an instructor can use them to help students see connections within the discipline that transcend individual course boundaries.

In some cases, it will be necessary to revisit the same Threshold Concepts over multiple courses. We saw evidence of this for both OOP and pointers. In object-oriented programming, we saw a student's understanding develop from seeing objects as a simple encapsulation mechanism to an appreciation for design patterns; for pointers (and the associated memory issues), the basic understanding learned in introductory programming became richer in subsequent courses. The students reported that crossing the thresholds for them was a gradual process, not necessarily an "aha" moment.

One more observation from the interviews that applies more generally than to Threshold Concepts, is that students (in retrospect) appreciate the value of individual work. Nearly all, when asked to give advice to other students who are having problems with a particular concept, included things like "you need to work on your own", "you need to sit down and think about what you're doing", and "sit down and work it out on paper and really understand what happens." Our own teaching experience is that this understanding is not shared by all novice students.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we describe Threshold Concepts. These are a subset of the core concepts in a discipline that are transformative, integrative, irreversible, and potentially trouble-

some. If instructors don't take care in introducing these concepts to our students and monitoring their understanding, students may fail to move on, or move on with only surface knowledge of the concept.

We also describe our initial empirical investigations into Threshold Concepts. We used both informal interviews and questionnaires to obtain data from instructors and scripted, semi-structured interviews to obtain data from students.

Based on these investigations, we present evidence that Threshold Concepts do exist in computer science. We have identified two Threshold Concepts, or perhaps broad areas within which thresholds exist: pointers and object-oriented programming. These were the terms our subjects used for concepts they identified, but these concepts – object-oriented programming in particular – are very broad. A close reading of the interviews suggests that more specific Threshold Concepts might include the way in which objects work together (i.e., concurrency), or the ability to see large problems as composed of a set of small sub-problems.

In future work, we plan to investigate these more specific concepts, and also some of the other candidate Threshold Concepts that have been mentioned in our data. One particularly intriguing example is the notion of translation from one representation to another – it is certainly pervasive within computer science, but only one of our instructor subjects (and none of the students) mentioned it as a candidate.

In addition, we have not yet analyzed the variations in understanding associated with the concepts discussed here. This analysis will provide an outcome space with qualitatively different ways to understand a certain concept (as in [11]), information that should be immediately useful to instructors.

An area that we will be investigating is whether the learning of a Threshold Concept is an identifiable stage that all learners go through, or whether it is more of an individual phenomenon. Some of our data suggest that the Threshold Concepts that a student identifies are strongly influenced by his or her learning experiences–e.g., the only two subjects who suggested "How a processor control unit works" had taken a course where they were required to design one. In addition, it is possible that a Threshold Concept may seem to be universal if the concept is overly broad, as mentioned above regarding to object-oriented programming: subjects may agree on the broad concept, but this may be due to their experiencing different more-specific concepts as thresholds. We will address these issues by undertaking interviews with a finer-grained focus on particular concepts.

We are also planning to interview novices, to see how their perspective compares with the graduating students'. Finally, once Threshold Concepts have been precisely identified, the next step will be to design curricula and assessment tools to help student cross these thresholds more easily.

6. REFERENCES

[1] ACM/IEEE-CS Joint Curriculum Task Force. Computing curriculum 1991. Report of the IEEE Computer Society and ACM, 1990.

[2] A. Biermann. *Great Ideas in Computer Science: a gentle introduction*. MIT Press, 1990.

[3] J. G. Brookshear. *Computer Science: an overview*. Addison Wesley, sixth edition, 2000.

[4] P. Davies. Threshold concepts: how can we recognise them? 2003. Paper presented at EARLI conference, Padova. http://www.staffs.ac.uk/schools/business/iepr/docs/etcworkingpaper(1).doc (accessed 25 August 2006).

[5] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Putting threshold concepts into context in computer science education. In *ITiCSE-06*, pages 103–107, Bologna, Italy, June 2006.

[6] A. Eckerdal and M. Thuné. Novice Java programmers' conceptions of "object" and "class", and variation theory. In *ITiCSE-05*, pages 89–93, 2005.

[7] Joint Task Force on Computing Curricula. Computing Curriculum 2001, computer science volume. Report of the IEEE Computer Society and ACM, 2001. http://www.sigcse.org/cc2001/ (accessed 25 August 2006).

[8] M. Kölling. The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-Oriented Programming*, January 1999.

[9] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. Early programming: A study of the difficulties of novice programmers. In *ITiCSE-05*, 2005.

[10] P. A. Luker. There's more to OOP than syntax. *SIGCSE Bull.*, 26(1):56–60, 1994.

[11] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Ass., Mahwah, NJ, 1997.

[12] R. McCartney and K. Sanders. What are the "threshold concepts" in computer science? In T. Salakoski and T. Mäntylä, editors, *Proceedings of the Koli Calling 2005 Conference on Computer Science Education*, page 185, November, 2005.

[13] J. H. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49:373–388, 2005.

[14] J. H. F. Meyer and M. Shanahan. The troublesome nature of a threshold concept in economics. 2003. Paper presented at EARLI conference, Padova. (As reported in [13]).

[15] D. Perkins. The many faces of constructivism. *Educational Leadership*, 57(3):6–11, 1999.

[16] E. Roberts. The dream of a common language: The search for simplicity and stability in computer science education. *SIGCSE Bull.*, 36(1):115–119, 2004.

[17] G. M. Schneider and J. L. Gersting. *An Invitation to Computer Science*. Brooks Cole, second edition, 1998.

[18] A. Schwill. Fundamental ideas of computer science. *Bull. European Assoc. for Theoretical Computer Science*, 53:274–295, 1994.

[19] L. Thomas, M. Ratcliffe, and B. Thomasson. Scaffolding with object diagrams in first year programming classes: Some unexpected results. In *SIGCSE-04*, 2004.

Paper IV

European Journal of Engineering Education

RESEARCH ARTICLE

Variation Theory Applied to Students' Conceptions of Computer Programming

Michael Thuné[*a] and Anna Eckerdal [b]

[a,b]*Department of Information Technology, Uppsala University, Uppsala, Sweden*

Abstract

The present work has its focus on university level engineering education students that do not intend to major in computer science but still have to take a mandatory programming course. Phenomenography and Variation Theory are applied to empirical data from a study of students' conceptions of computer programming. A phenomenographic outcome space is presented, with five qualitatively different categories of description of students' ways of seeing computer programming. Moreover, dimensions of variation related to these categories are identified. Based on this discussion it is suggested how to use patterns of variation in order to support students' learning of computer programming. Finally, results from a pilot study demonstrate successful application of two patterns of variation in a computer lab assignment.

Keywords: variation theory; pattern of variation; phenomenography; computer programming

[*] Corresponding author Email: michael.thune@it.uu.se

1. Introduction

Computer programming is a mandatory subject in many university level engineering education programs. Students experience this to be a difficult subject. It was confirmed in a big multi-national study that university students world-wide have difficulties to master computer programming (McCracken et al. 2001).

Against this background, there is an urgent need to find ways to improve basic education in programming, for better student learning outcome. Research in this direction has mainly focussed on students' observed abilities to understand and master various elements of computer programming. For a recent survey, see Robins et al. (2003). In those studies, the overall meaning of programming was taken for granted.

In the present paper we look at the problem from a different angle, by addressing the following questions:

- *What are novice students' conceptions of computer programming?*
- *How can we design learning activities that will support students' learning by opening possibilities for them to see additional features of computer programming?*

To address the first question, we interviewed university students at the end of an introductory programming course. As a result we describe the variation in how those students express their experiences of what it means to program. Here, we will formulate this variation as a limited number of qualitatively different categories of description. To address the second question we then apply variation theory to those categories of description. The goal is to bridge the gap between students' conceptions and teachers' professional understandings of programming.

2. Variation as a key to learning and teaching

Our analysis uses the *Variation Theory* of learning and teaching (Marton and Booth 1997, Marton and Tsui, 2004). First, there is a qualitative variation in the ways in which phenomena —related to the object of learning—are experienced by learners. This observation is the basis for *Phenomenography*, an empirically based, qualitative research approach. A phenomenographic study aims to describe the variation in the ways people experience or 'see' a phenomenon. For examples and references, see Marton and Booth (1997). The data for a phenomenographic study are typically collected via interviews with individuals. The data are analysed on a collective level, with the aim to find all the *qualitatively different* ways of seeing the phenomenon that are expressed by the interviewees. These qualitatively different ways of seeing are finally formulated by the researchers as analytical *categories of description* that form the *outcome space* of the phenomenographic analysis.

The outcome space will usually have a hierarchical structure, going from categories including few features of the phenomenon, to categories describing richer or deeper ways of seeing, encompassing several features. The crucial question for the educator is how to help students to develop rich ways of seeing the phenomenon.

Discernment and *variation* are key words to answer that question. According to Variation theory a necessary—but not sufficient—prerequisite for discerning a feature, is to get the opportunity to experience variation in a *dimension* corresponding to that feature. For example, if 'size' and 'colour' are the features of a phenomenon 'picture component', then there is a 'size' dimension and a 'colour' dimension of the corresponding feature space. A particular instance of 'picture component' can be represented by its values in those dimensions, i.e., by its particular size and colour.

In summary, via interviews we can record how students express their ways of seeing a phenomenon. As a second step, a phenomenographic analysis of the interviews will lead to the formulation of categories of description of a limited number of qualitatively different ways in

which students see the phenomenon. Finally, given these categories, it will be possible to look for dimensions of variation and to discuss how to bring the variation about that would enable students to develop richer ways of seeing the phenomenon. Below, we go through these three steps for the particular phenomenon of interest in the present paper, i.e., computer programming.

3. The data collection

Our study is based on interviews with first-year students enrolled in a study programme in Aquatic and Environmental Engineering at Uppsala University, Sweden. This is a prestigious, highly theoretical four and a half year programme, leading to an M. Sc. degree. After graduation the students generally get employment as qualified specialists in their field of expertise. They are a typical example of a group of university level engineering students who are supposed to learn the basics of computer programming, but are not expected to head for future careers as programmers.

The interviews were semi-structured (Kvale 1996). This means that on one hand there was a structure in the form of 'lead questions' to open up different themes to be covered but on the other hand each lead question was open, to give the interviewee opportunities to freely express his/her understandings and experiences. The interviewer asked follow-up questions to encourage the students to talk about their experiences from different perspectives and with increasing preciseness. The interviews were conducted when the students were at the end of the introductory programming course, addressing object-oriented programming and the programming language Java. Fourteen students were interviewed. Since the objective was for the interviews to exhibit the variation in the student group, the interviewees were selected with the aim that they collectively should cover as broad a range of relevant characteristics as possible: age, gender, and previous educational, professional, and hobby experiences of computer programming. The length of an interview was typically around one hour. Each interview was recorded on tape, and then transcribed verbatim. The analysis of the data is based on the transcripts.

4. Interpretation of students' experiences

We subjected the complete set of transcribed data to a phenomenographic analysis. This is an iterative process, where the transcribed interviews are read and re-read. Statements from different interviews indicating similar ways of seeing are extracted from their contexts and grouped together, to form tentative categories of ways of seeing.

The final result of our phenomenographic analysis is a set of five categories of description of students' qualitatively different ways of seeing computer programming. These categories of description are briefly summarised in Table 1. Note that each category of description is based on a group of excerpts extracted from different interviews. A comprehensive account of the categories of description and their empirical basis is beyond the space limitations for the present article. The interested reader is referred to Eckerdal (2006) and Eckerdal and Berglund (2005), where students' conceptions of *learning* to program were in focus, based on the same pool of empirical data. The categories of description reported there are very similar to the ones found in the present analysis, since students' conceptions of learning to program turned out to be strongly related to their conceptions of computer programming.

The five categories of description are organised in a hierarchy, where the categories are enumerated in increasing order of richness or depth. Each category presumes the understanding

expressed in the preceding categories of description and is qualitatively different from these by including an additional feature of computer programming.

Category 1 describes an understanding that is directed towards the programming language, to understand its syntax, to know its key words by heart. For example, Student N[1] says:

> N: What it is all about, I think it is all about learning, partly the commands, fundamental commands I use, I have to remember them [...]

In Category 2 there is awareness of the relation between the program text and the actions that take place when the program is executed. According to the understanding described in Category 2, the correspondence between text and action is difficult to capture and programming is regarded as an almost mystical way of thinking that can only be mastered by those who have understood this relation. Student D says:

> D: […] how one should use different things in a program […] it feels as some people just understand programming, it's something they... but I also think that some people who have been programming before have probably learned to think like that.

In Category 3, the applications of programming are also included. The following statement by student B illustrates this:

> B: […] for me it has been about how a lot of things are built up. That is to say that I've understood a lot of things, how bank programs work and such.

The understanding described in Category 4 includes the same features of programming as the previous three categories, but now the understanding of the relation between text and action is clearer and the thinking required to program is seen as a kind of systematic problem solving. Student C says:

> C: […] I guess it's actually to solve a certain type of problem, it's rather like the math courses. Then, learning different methods to solve them in different ways.

Finally, Category 5 includes all the previous categories and adds the insight that computer programming can be used outside the programming course and for other purposes than programming. One aspect of this is expressed by Student E:

> E: Yes but it's that the more you know about computers the less dependent on others you'll be, sort of.

Similar questions as the ones posed in our study, have been addressed by other researchers in the phenomenographic tradition. In particular, Table 1 points in the same direction as the outcome spaces reported by Booth (1992), and Bruce et al. (2004), respectively. As part of a larger study, Booth investigated students' conceptions of learning to program. Her outcome space has four categories, three of which are similar to our Categories 1, 2 and 4. Her fourth category, 'learning to program as becoming part of the programming community' does not appear in our data. Most likely, this is due to the fact that Booth interviewed computer science majors, while the students in our study were not heading for careers in the software industry. This difference is probably also the reason why our Category 3 did not appear in Booth's outcome space. Similar comments can be made with regard to the study by Bruce et al. (2004). We do not go into a detailed comparison here. The important message is that Table 1 points in the same direction as the outcome spaces by Booth, and Bruce et al., respectively. These three outcome spaces paint the same basic picture, while each outcome space adds partly different nuances to it. The additional contribution made below consists in a further analysis of the outcome space summarised in Table 1, with the aim to further explore the dimensions of variation related to it.

[1] To protect the anonymity of the participating students, each student is referred to by a letter which is completely unrelated to the student's real name.

Table 1 represents the lived object of learning (Marton et al. 2004), that is, how the students actually were able to see computer programming after having taken an introductory university course on that subject. From an educational point of view, it is of interest how those categories of understanding relate to the intended object of learning (Marton et al. 2004), as conceived by educators. An authoritative source of information concerning the intended object of learning in introductory computer programming courses is provided by *Computing Curricula 2001—Computer Science* (CC 2001, see Engel and Roberts 2001). It is implied in CC 2001, Chapter 7.2, that a desirable goal for an introductory programming course is that students attain the ability to "adapt to different kinds of problems and problem-solving contexts" in the future. This corresponds fairly well to the understanding of computer programming described in Category 5, keeping in mind the inclusive character of the five categories of description.

5. Variation for Discernment

The outcome space in Table 1 demonstrates a wide span between different novice programmers who have taken the same introductory programming course. The phenomenographic analysis addresses the collective level, and it is not possible to directly map individuals to categories.

We nevertheless argue that a majority of our interviewees had not reached the more advanced ways of seeing represented by the richer categories of description in Table 1. Although the interviewer tried to bring in new perspectives via follow-up questions, many students did not answer to that by associating to new features of the phenomenon.

The challenge is to create learning conditions that enable students to see additional features of computer programming. According to variation theory, creating such learning conditions means to create possibilities for students to experience variation in dimensions related to the features. Experiencing such variation is a necessary, though not sufficient condition for discerning the features.

We will now examine the categories summarised in Table 1, in order to distinguish dimensions of variation that a teacher in an introductory programming course needs to address. In Category 1, focus is on one particular feature of computer programming: that programming includes writing program *text*. In that sense, the textual representation of programs constitutes a dimension of variation of relevance for the understanding of computer programming. Different, specific program texts provide values in this dimension.

In order to experience computer programming according to Category 2, the student also needs to focus on what happens when the program is executed. Consequently, the dimension of program action during execution is of relevance for understanding 'computer programming' as described in this category. Different specific cases of programs, with different actions, provide values in this dimension of variation.

Category 3 brings in an additional awareness, namely that programs occur in everyday life. The dimension of variation of relevance in this context is the applications of computer programs. Real-life application programs for different applications provide values in this dimension.

In Category 4, the problem solving aspect of programming comes into focus. The dimension of relevance, then, is constituted by application *problems* to be solved by means of computer programs. Different problems provide values in this dimension of variation.

Finally, Category 5 represents students' expressions of experiencing computer programming as something more general than to program computers. Programming is experienced as a skill that the student can benefit by in the future, and that will be empowering. The corresponding dimension of variation is 'The various contexts in which programming skills can be an empowering resource'.

6. How to bring variation about

In summary, the previous discussion led us to identify five dimensions of variation that a student needs to discern in order to see computer programming in the way that is reflected by Category 5 in Table 1, the category that is closest to the intended object of learning. These dimensions are summarised in Table 2.

The educational implication is that the teacher should take deliberate measures to create opportunities for the students to become focally aware of all dimensions in Table 2. Variation theory says that a necessary condition for a student to discern a certain dimension—and the corresponding feature—is that the student gets the opportunity to experience variation in that dimension. How can this be done?

There is no unique answer to this question. Any experienced programming teacher can think of different ways of exposing the required variation. On the other hand, it is apparent from our study that it is not sufficient with just *any* variation. In programming education it is common teacher practice to show many program examples to the students. All the above dimensions of variation would be possible to observe by comparing different examples. Nevertheless, far from all students discern them. This is in line with the findings from a large number of classroom studies where different kinds of variation were tried, in various subject areas at the high school level (Marton and Tsui, 2004). A general conclusion from these experiments is that "it is necessary to pay close attention to what varies and what is invariant in a learning situation, in order to understand what it is possible to learn in that situation" (Marton et al. 2004).

The key point is that a *combination* of variation and invariance will be required in order to create favourable conditions for students' to discern a certain dimension. Marton et al. (2004) introduce the concept of *patterns of variation* to refer to ways of systematically combining variation and invariance. They mention four such patterns that they have identified: *contrast*, *generalization*, *separation*, and *fusion*, respectively. Briefly, the *contrast* pattern means to contrast a phenomenon *P* to other related phenomena, to make it possible to discern *P* as a phenomenon distinct from other phenomena. The *generalization* pattern means to exhibit varying specific appearances of *P*, in order to open the possibility to discern the general meaning of *P*. The *separation* pattern means that there is variation in precisely one dimension, to create the possibility to discern that particular dimension, keeping the other dimensions invariant. Finally, the *fusion* pattern means to exhibit variation in several dimensions simultaneously, to open the possibility to discern the relations between these dimensions.

7. Variation in programming education

We will now give *examples* to suggest how the general ideas discussed above could be applied in a novice programming course. In Section 7 we will briefly mention some ways to bring variation about in programming education. A couple of the suggestions will then be further concretised in Section 8.

Let us first consider how to create possibilities for students to become focally aware of the second dimension listed in Table 2, i.e., "program action". The differences in action between two programs will become more striking—and thus more likely to be perceived by students—if the textual difference between the programs is very small. We suggest the notion of *pseudo separation* for this particular form of the fusion pattern of variation. In our example, the purpose of pseudo separation would be to give the students a possibility to discern program action as a separate dimension of programs and programming.

Once the students have discerned this dimension, the proper fusion pattern can be used for opening the possibilities to explore the *relation* between program text and program action. This can be supported by using software development tools that allow for simultaneous observation of source code and program action at run time.

Next, we discuss the fourth dimension listed in Table 2. How can we use patterns of variation to open possibilities for students to discern the tasks presented to them in the programming course as problems. We propose to use the generalization pattern to this end. The idea would be to expose the students to different kinds of problem descriptions, and to include among those some descriptions where the problem is to construct a computer program for an application. This can open the possibility for students to see the similarity between problem formulations in a programming context and other problem descriptions, and that all these descriptions are related to the concept of 'problem'.

To further open the possibilities for students to experience programming according to the description of Category 4 in Table 1, it is advisable that *some* systematic approach to program development is introduced to the students. In Chapter 7.2 of CC 2001 (Engel and Roberts 2001), it is stated that the traditional introductory programming course gives "too little weight to design, analysis and testing" and that "concentrating on the mechanistic details of programming constructs often leaves students to figure out the essential character of programming through an *ad hoc* process of trial and error". To make it possible for novice programming students to see the significance of addressing problem solving in a systematic way, with problem analysis and program design, the contrast pattern of variation might be useful. An exercise where they were prompted to solve the same programming task twice, using first an ad hoc approach and next a systematic one, could make it easier to see the distinguishing characteristics of a systematic approach. Here, "systematic" should be interpreted from the novice programmer's perspective: making an elementary analysis of the problem and sketching a program design before beginning to write program text.

8. A pilot study

We carried out a small pilot study to test two of the suggestions made in the previous subsection. A computer lab assignment was designed to include the pseudo separation pattern and the fusion pattern of variation, to give students an opportunity to become aware of the second dimension in Table 2—"the action of a program"—and to get an understanding of the relation between the two first dimensions in that table, "textual representation" and "action".

During the scheduled, teacher-supervised lab session, the students worked with a ready-made Java class `Radio`, representing an fm radio. The pseudo separation pattern was used in one part of the lab session. The students had access to a program named `RadioTest`. They were required to make changes in that program and to observe the effects on program output. For example, they should change the statement `r1.switchChannel(2,time)` to `r1.switchChannel(1,time)`. This minor change of the program text had a significant effect on program action, namely that the radio `r1` tuned in to a news channel instead of an opera music channel. We hoped that this would help students to experience program text and program action as two separate dimensions.

The lab also contained a task designed according to the fusion pattern. The students were required to use a debugger when they executed `RadioTest`.[2] This opened the possibility for the students to get a better understanding of the relation between program text and program action by seeing in detail how each instruction in the program text affected the values of program data at execution.

[2] Explanation for non-CS experts: The debugger is a software tool that allows for step by step execution of programs. After each step it is possible to examine the values of the data stored in the program's working memory.

To assess the effect, we asked the students to write a short reflective essay after the lab session. The question to answer was: "How has your understanding of programming developed as a result of what you learnt during the lab session?" Eventually, 22 students handed in an essay.

A majority of these, 14 students, reported on improved understanding. Notably, several students expressed that the lab had helped them to get a better understanding of the relation between the program text and what happens when the program is executed.

For example, one student reported: "After today's lab session...by reading ready-made source code I understand better what the program will do in detail". Another student wrote: "It felt like the first time one really exercised power over what the program did". As a third example, one student experienced that "the [method] calls and how everything connects became a bit clearer during the lab session".

The remaining 8 students claimed not to have benefited from the lab session.

In conclusion, the results of our pilot study indicate that the lab assignments opened a space of learning where the two dimensions "program text" and "program action" could be perceived, as well as relations between these dimensions. The majority of the students in our pilot study experienced that the lab had improved their understanding. In addition, as demonstrated by the quotations above several students phrased their learning experience in words indicating that they actually came to see the two dimensions and their relations better as an effect of the lab.

9. Conclusions

To get a nuanced understanding of novice students' conceptions of computer programming, we collected data via interviews at the end of an introductory programming course. A phenomenographic analysis of these data resulted in the categories of description summarised in Table 1.

In a second step, we interpreted the categories in Table 1, to identify different dimensions of variation. Table 2 summarises the results of this step.

Finally, we discussed how the patterns of variation identified by Marton et al. (2004) could be used by teachers of computer programming, in order to open possibilities for students to discern the dimensions described in Table 2. The results of the pilot study reported above are encouraging.

In conclusion, variation theory provides insight that can be used by educators for reflecting upon their own practice. First, Table 2 is valuable insight per se, since it helps the teacher to focus attention more consciously on those particular dimensions.

Secondly, variation theory can be used as a starting point for teachers' design of educational situations. There is a repertoire of patterns of variation that teachers can select from, and adapt to their own contexts and preferences. Here, we have suggested *some* ways of using patterns of variation in programming education. The pilot study reported above, with carefully planned variation in the lab, gave encouraging results. The lab clearly helped students to see important features of programming and in several cases the learning experiences reported by the students were in agreement with the intended goals for the variation theoretically based design of the lab assignments. To establish to what extent the variation actually contributed to the learning taking place, an interesting next step would be to go beyond the pilot study and conduct a more comprehensive empirical test of these ideas in the form of a Learning Study (cf. Lo et al. 2004). This line of future research can lead to important insights about programming education.

References

Booth, S. A., 1992. *Learning to program.* Thesis(PhD). Gothenburg University, Sweden.

Bruce, C., Buckingham, L. Hynd, J., McMahon, C., Roggenkamp, M., and Stoodley, I. 2004. Ways of experiencing the act of learning to program. *Journal of Information Technology Education*, 3, 143–160.

Eckerdal, A. and Berglund, A., 2005. What Does It Take to Learn 'Programming Thinking'? *In: Proceedings of the 1st International Computing Education Research Workshop*. New York: ACM Press,135–143.

Engel, G. and Roberts, E., eds., 2001. Final Report on Computing Curricula 2001—Computer Science. *ACM Journal of Educational Resources in Computing*, 1(3)

Lo, M. L., Marton, F., Pang, M. F., and Pong, W. Y., 2004. Toward a pedagogy of learning. *In:* Marton and Tsui (2004), 189–225.

Marton, F. and Booth, S. A., 1997. *Learning and awareness*. Mahwah, NJ: Lawrence Erlbaum Associates.

Marton, F., Runesson, U., and Tsui, A. B. M., 2004. The space of learning. *In:* Marton and Tsui (2004), 3–40.

Marton, F. and Tsui, A. B. M., eds., 2004. *Classroom discourse and the space of learning.* Mahwah, NJ: Lawrence Erlbaum Associates.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., 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. *SIGCSE Bulletin* 33(4), 125–180.

Robins, A., Rountree, J., and Rountree, N., 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137–172.

1. Computer programming is experienced as to use some programming language for writing program texts.
2. Computer programming is seen a way of thinking that relates instructions in the programming language to what will happen when the program is executed.
3. Computer programming is seen as a way of thinking, as above, and in addition computer programming is experienced as producing computer programs such as those that appear in everyday life.
4. Computer programming is seen as described above with the addition that computer programming is experienced as a "method" of reasoning that enables problem solving.
5. Computer programming is seen as a way of thinking, to solver problems, leading to the production of computer programs such as those that appear in everyday life. In addition computer programming is experienced as a skill that can be used outside the programming course, and for other purposes than computer programming.

Table 1: Summary of categories of description of students' qualitatively different ways of experiencing computer programming

The textual representation of a program
The action of a program
The application addressed by a program
The problem to which the program is a solution
The various contexts in which programming skills can be an empowering resource

Table 2: Five dimensions of variation related to computer programming.

Paper V

What Does It Take to Learn 'Programming Thinking'?

Anna Eckerdal
Department of Information Technology
Uppsala University
P.O. Box 337, 751 05 Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Anders Berglund
Department of Information Technology
Uppsala University
P.O. Box 337, 751 05 Uppsala, Sweden
Anders.Berglund@it.uu.se

ABSTRACT

What is 'programming thinking'? In a study, first year students were interviewed on their understanding of what learning to program means. Many students talked about learning to program in terms of learning a special way to think, different from other subjects studied. Many of these students had problems in describing what this special way to think included. The analysis of the interviews revealed some features of this thinking, as expressed by the students. In this paper we discuss and analyse 'programming thinking' using phenomenography as our research approach [7]. Our results are coherent with Hazzan's research on the learning theory 'process-object duality' [4], but points to problems in learning of object-oriented programming not indicated in 'process-object duality'. In comparing the results form our own study with this learning theory, we discuss what this might mean in learning object-oriented programming.

Categories and Subject Descriptors

K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Computer science education*; D.1.5 [**PROGRAMMING TECHNIQUES**]: Object-oriented Programming

General Terms

Human Factors, Theory

Keywords

Phenomenography, process-object duality, levels of abstraction

1. INTRODUCTION

What is 'programming thinking'? When interviewing students about their experiences of learning to program, many of them expressed that programming includes a different way of thinking. This 'programming thinking', as some of the students said, seemed to have an almost magic character. The students had difficulties to describe and get a grip on how to apply this kind of thinking. The present paper discusses this and related issues.

In our study, first year students taking an object-oriented programming course were interviewed on what it means to learn to program. Through our analysis we found that there are certain levels of understanding that seemed necessary for reaching a level of abstraction where concepts can be used for analysis and design in object-oriented programming tasks. We argue that some understanding scaffolds this level of abstraction, and that there are ways for educators to facilitate for the students to reach such understanding. These may include problem solving and standard methods in programming.

The understanding gained from this study is important, since object-oriented programming languages are used in university courses at all levels throughout the world. Much has been reported on the experiences of teaching on the object-oriented paradigm. Object-oriented programming is experienced as complex and difficult to teach [5] [8]. One important aspect of learning the object-oriented paradigm is that it is built on some fundamental abstract concepts, and there is a need for students to reach certain levels in their understanding of these concepts.

Section 2 gives a theoretical background for the study and the analysis performed. The study and the results are then presented in section 3. In section 4 we relate our results to previous pedagogical research [4] and discuss implication for teaching. We show that our results and the pedagogical research illustrate the same progression in learning of abstract concepts. The current project, as well as the work of Hazzan, report about the same phenomenon, but using different approaches. This strengthen the conclusions. Our results however put a searchlight on problems not indicated in the research mentioned. There are students who seem to have problem to discern the methods and procedures available when solving problems in object-oriented programming. We argue that these procedures can act as bridges to more abstract understanding of concepts in object-oriented programming. It is thus important for educators to facilitate for students to discern such procedures. This discussion is closely related to the question *what does the students' experience of 'programming thinking' mean when learning object-oriented programming?* which is the main focus in the study presented here.

ICER'05, October 1–2, 2005, Seattle, Washington, USA.

2. RESEARCH APPROACH

2.1 Phenomenography

Phenomenography aims at describing the variation of understanding of a particular phenomenon found in a group of people. Marton and Booth discuss [7] the idea of phenomenography:

> The unit of phenomenographic research is *a way of experiencing something*, [...], and the object of the research is the *variation* in ways of experiencing phenomena. At the root of phenomenography lies an interest in describing the phenomena in the world as other see them, and in revealing and describing the variation therein, especially in an educational context [...]. This implies an interest in the variation and change in capabilities for experiencing the world, or rather in capabilities for experiencing particular phenomena in the world in certain ways. These capabilities can, as a rule, be hierarchically ordered. Some capabilities can, from a point of view adopted in each case, be seen as more advanced, more complex, or more powerful than other capabilities. Differences between them are educationally critical differences, and changes between them we consider to be the most important kind of learning. [7, p. 111]

And later:

> [...] the *variation* in ways people experience phenomena in their world is a prime interest for phenomenographic studies, and phenomenographers aim to describe that variation. They seek the totality of ways in which people experience, or are capable of experiencing, the object of interest and interpret it in terms of distinctly different categories that capture the essence of the variation, a set of categories of description [...] [7, p. 121-122]

The object of interest in a phenomenographic study is thus how a certain phenomenon is experienced by a certain group of people. An empirically based insight in phenomenography is that there is a limited number of qualitatively different ways in which a certain phenomenon can be understood.

Phenomenography is an empirical, qualitative research approach, often used in educational settings. Data can, as in the present study, be gathered in the form of interviews. The interviews are transcribed and analysed. Researchers, in this case two, analyse the data in order to find qualitatively different ways subjects understand the phenomenon uncovered. The researcher formulates the essence of the understanding as categories of description. It is important to state that the analysis is on a collective level, not individual students' understandings. This is done by reading and rereading the interviews, in context, but also by decontextualising excerpts, comparing them and grouping them together in different categories. The resulting description of qualitatively different categories of understanding constitutes the *outcome space* of a phenomenographic analysis. In this way we thus identify aspects of the understanding of the phenomenon, from the students' perspective.

3. THE STUDY

3.1 The Interviews

A study has been performed where 14 students were selected for a one-hour tape-recorded interview. The students participated voluntarily in the study, but got one movie ticket each as a sign of recognition. The students enrolled were in a program called Aquatic and Environment Engineering. This educational program was selected because programming is not the main focus for the students, and the students are thus representative for a large number of students studying programming. The students had just finished their first programming course, a mandatory course giving 4 credit points. One credit point at Swedish universities corresponds to one week full time studies. The programming language used in the course was Java.

Most students taking part in the course filled in a questionnaire about previous programming knowledge, education, work experiences and gender. On the basis of these answers, we selected interviewees that represented as broad a coverage as possible of the factors mentioned.

The interviews were semi-structured [6] with a small number of pre-prepared questions, intending to approach the phenomena of interest in different ways to give the opportunity for the students to express as much of their understanding as possible. The primary question asked to the students was:

- What do you think learning means (involves) in this course?

Other questions were:

- What do you experience this course to be about?
- What has been most important to you in this course/ Why has this course been good for you?
- What do you think was the aim for you when learning to program?
- What has been difficult in the course?

3.2 The Phenomenographic Analysis

One researcher read and analysed the transcribed interviews, looking for qualitatively different ways to understand the phenomenon *what does it mean to learn to program* expressed in the data. The second researcher studied quotes from the students and the categories identified by the first researcher. Five different ways to understand the phenomenon found in the data were agreed upon. (A discussion on trustworthiness in phenomenography can be found in [1].)

These categories are presented in Table 1. The categories are inclusive. This means that an understanding expressed in one of the later categories includes the understanding expressed in the former categories. The categories are furthermore hierarchical in the sense that the new understanding expressed in the later categories are more advanced.

Three of these are directed towards the computer, the programming language, and programming in general, while two are directed outwards, towards the society with its programmed artifacts and the world of the programmer. Each category is described and illustrated with excerpts from interviews. In the quotes, the interviewer is labeled I, and the students A, B, C etc. The table and the comments serve as a background for the discussion in this paper. We will focus on categories two and four. The results are then compared with the learning theory 'process-object duality' from mathematics. A more elaborated work on the results on students' understanding of what it means to learn to program,

1. Learning to program is experienced as to understand some programming language, and to use it for writing program texts.
2. As above, and in addition learning to program is experienced as learning a way of thinking, which is experienced to be difficult to capture, and which is understood to be aligned with the programming language.
3. As above, and in addition learning to program is experienced as to gain understanding of computer programs as they appear in everyday life.
4. As above, with the difference that learning to program is experienced as learning a way of thinking which enables problem solving, and which is experienced as a "method" of thinking.
5. As above, and in addition learning to program is experienced as learning a skill that can be used outside the programming course.

Table 1: Categories describing the students' different understanding of the phenomenon *What does it mean to learn to program?*.

including the other categories in Table 1, will be presented in later work.

Before describing these categories, let us note that students' experiences of what it means to learn to program has been investigated in some previous studies [2, 3]. In particular, category two and three have not appeared explicitly in the previous studies. Since our focus is on category two and four, the other categories will be only briefly described in this paper.

3.2.1 *Learning is to understand some programming language, and to use it for writing program texts*

The first category summarizes an understanding that is directed towards the programming language itself, to understand it and to be able to use it. It is expressed in such a way that students' description of learning the details of syntax gives the feeling of knowing how to program. Other students focus on the ability to write short pieces of programs, 'program chunks', as characterising what learning means. To sit by yourself and code is desirable and appreciated. The skill to code in Java and to remember details in the language summarize this understanding. This understanding is presupposed in the other understandings the students express.

Student N emphasizes the importance of detail knowledge of the syntax, and learning by heart. Answering the question what it means to learn in this course, student N answers:

> N: (giggle) Yes, but to learn must mean to understand and... but it doesn't mean that, because we have done the mandatory assignments in pairs, so it doesn't mean to sit beside and look when the other person does it, of course. Yes but to pick up what it's about, to understand what it's about and hopefully remember something.
> I: What is it about then?
> N: Well (giggle), don't know... difficult to say.
> I: [...] what is your opinion of what it is all about?
> [...]
> N: What it is all about, I think it is all about learning, partly the commands, fundamental commands I use, I have to remember them [...]

Student B emphasizes learning by coding on your own. The aim with the studies is expressed as being able to code on your own without any help:

> I: What do you think it means to learn in this course?
> B: It probably means that I'll be able to sit and do small simple things on the computer by my self. That I've learned that I think. [...] Eh, but otherwise it's that I'll know the foundations and that I'll kind of be able to try myself I suppose.

3.2.2 *Learning is a way of thinking, which is experienced as difficult to capture, and which is understood to be aligned with the programming language*

A common way to express what it means to learn to program, or what is missing in their understanding of programming, can be described as 'programming thinking'. Approximatly half of the students in the study talk about the actual thinking behind programming as something specific, an ability one has to acquire to be able to program. Many of these students seem to have problems identifying what 'programming thinking' involves. Some express themselves as if it is something magic, difficult to catch.

Category two in Table 1, *Learning to program is experienced as learning a way of thinking, which is experienced as difficult to capture, and which is understood to be aligned with the programming language*, summarizes this understanding. This second category includes the first one, *Learning to program is experienced as to understand some programming language, and to use it*, but is more developed. An example that illustrates that the first category is included in the second is when student D says: "you're supposed to get an understanding of the actual thinking when you program." The student discusses programming, but the focus is on the special thinking that is required. The understanding is, like in the first category directed toward the programming language but also toward the logic and thoughts behind the language. In this spirit some students discuss the differences between human beings and computers as something crucial to grasp in the learning of programming. Student C talks about what is most important in the course:

> C: ...it's probably the way of thinking, that is when you program, how you are supposed to think and computer code and how it is interpreted, that's the difference to how human beings think.

Some students use the word 'logic' when they discuss how to think when they learn to program. This 'logic' is discussed by student A when he/she is asked what is most important in the course:

> A: It is the understanding of how the programming language is built rather than the specific command, if you want to do this, it's more the thinking itself, the logical thinking. Everything you need to know you must think of when it comes to programming. It's kind of, yes, it's very

> exclusive, everything is simply very detailed and you've kind of got a small insight into what it's like to program and how the computer works like that, or the software.

Student A articulates that the problems with logic are the precise demands of the syntax of the programming language. Student A also connects this special thinking to how the computer itself works, not only the features of the programming language. It is interesting to compare this with another statement from student A:

> A: Yes, okey. Well, it's a little bit interesting. I can, you kind of get an eye-opener, [...] when you kind of sit with the computer, that you realise how terribly much you can do, that you can do yourself with the help of a keyboard and this I have got an understanding of in the same way as I really understand those who are kind of confirmed programmers and think this is so much fun.

Student E expresses 'programming thinking' as different choices to reach a specific goal. He/she answers the question what it means to learn in this course:

> E: Well it's like thinking programming I think. Understanding things, puting together and how you make things work sort of and accomplishing what you want yourself. There are also many roads to take yourself to the goal.

Student D, who is enrolled in the program Chemical Technology Engineering, describes this special way of thinking as making it difficult to know how to construct a program, to understand concepts, and that it also causes problems in knowing how to go about studying. Student D talks about programming thinking in a way reminiscent of magic. On the question what has been difficult in the course student D says:

> D: Yes, I think it has been difficult with concepts and stuff, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult, but this very thought behind, it feels as some people just understand programming, it's something they... but I also think that some people who have been programming before have probably learned to think like that. But I still think the course, it's difficult for a novice to sort of get a grip of how to study when you implement the programs and like that. (Giggle)...

Student D also discusses the reputation the course has among the students in his/her own program, and compares programming with other subjects studied.

> I: Where do you think the problems lay?
> D: I don't know. I guess, I think everyone has kind of a dread of programming (laughter) it's kind of, in my class...
> I: Really...
> D: ...no but... (laughter)
> I: ...more than math?
> D: Yes really, much worse. No but it's kind of, in my class...
> I: Chemistry is it there...
> D: Yes my class is chemistry, those who are doing the second year now, it's this very course that most of the students haven't passed. More than all courses. Everyone just goes, poor you, are you taking programming, like! (Laughter from both). So I've been scared by them.
> I: But you have studied chemistry.
> D: No then you get so scared and just Oohh. I guess, it's just a rather different way of thinking.
> I: Now that you have taken this programming course could you put your finger on something you think is different than than chemistry... or you must have taken math too I suppose.
> D: Sure, I've taken many math courses but math is kind of logical and you understand it but this is... no I don't know (laughter). No but I kind of think it's easier to study math. Then you often have something creative to base it on, or you don't, but you learn more methods and kind of, there is some theory behind. Here you feel as if you only learn a lot of examples. You know, we've gotten so many examples of everything, in some way it feels as if you don't understand the base from the beginning [...]

Student D experiences a lack of method and theory when learning to program compared to when learning mathematics. Mathematics is thus experienced as an easier subject to study.

Student C also makes a comparison between how to think in programming and how to think in other subjects:

> I: Yes. You mentioned something about ways to think.
> C: Mm. Yes, that, what should you say about this? But exactly this kind of, that you have to be so precise in everything you describe to the computer that you want it to do. It's not very open for interpretations. [...]
> I: Do you think it's a different way of thinking compared to what you've met in other courses?
> C: Yes we have mostly done math until now actually and there it's quite different. Then you think on you own, kind of, it's enough to have a way of thinking just for me. Now it must suit a way of programing as well, it's more like I get shaped into it, thinking like the computer, or like Java is written, but in math it's enough kind of that it works.
> I: Does it mean that you feel more free kind of to think in math?
> D: Yes I definitely think so.

3.2.3 *Learning is to gain understanding of computer programs as they appear in everyday life*

Some students talk about the programming they come across in everyday life. Category three in Table 1, *Learning to program is experienced as to gain understanding of computer programs as they appear in everyday life*, summarizes this understanding.

Student D answers the question what was most important in the course:

> D: [...] You just think of things like when you withdraw money from a cash point, kind of, then you start to think, okey, it's these steps, figures and the sum and kind of... if there is money in the account and so on. No but those things that one starts to think a little about how certain things are built and exactly, yes, such things as when you're going to withdraw money or different games or such.

Student C answers the same question:

> C: Yes, no I don't know. It probably will be useful perhaps now and then or the understanding of how devices work in general. And machines. [...] No but there are many things that are run by computers today undeniably so that, it's some kind of understanding how things work. It's in cars, computers, lifts and everything. So that, yes, no, a good overview.

The quotes from the students above express a vague and shallow understanding of programming as something they meet in everyday life that *might* be of some use, because of the wide spread of computer programs. The third category includes the first two categories in Table 1 because it discusses computer programs and the thinking used when building programs. Despite its superficialness it bridges to the last two categories found in the data when reaching out beyond the programming language and the course itself. The last two categories in Table 1 express understandings that are richer than the understandings expressed in the first three categories.

3.2.4 *Learning a way of thinking, which enables problem solving, and which is experienced as a "method" of thinking*

The category expressed in category four in Table 1, *learning a way of thinking, aligned with the programming language which enables problem solving*, talks about learning to program in terms of problem solving. It is closely related to the understandings expressed in categories one and two. Programming knowledge is more or less presupposed, and 'programming thinking' is connected either to the course and course context, or to a need not limited by the course itself with its specific language learned. By taking the discussion outside the course context the understanding expressed in category four reaches beyond the first two categories and includes and builds upon the third category which discusses programming as it is met in everyday life. 'Problem solving' is discussed as an ability useful both within the course and outside of the course context.

Student G discusses what it means to learn in the present course:

> G: To get to try, like, you learn to think in a special way, you learn problem solving. [...] It's problem solving. With the mandatory assignments, that is the difficult part, this you can say at least I think so.

Notice that student G mentions problem solving at the same time as he/she talks about learning a certain way to think. Problem solving is seen as part of 'programming thinking'.

Student K discusses problem solving as an ability separate from the programming language learned in the course. When answering the question what it means to learn in the present course student K says:

> K: [...] You know, it's good to have this kind of courses because you get to kind of exercise problem solving. That's actually really good. You have a problem that you solve in different ways and then you perhaps find the best way. That's one of the central parts I think. Then that you must write in some programming language, that you can perhaps do in any language. But exactly the problem solving, the way to handle problem solving, that's what I important think is important.

Student C answers the same question. He/she focuses on problem solving as meaning certain types of problems appearing in the course. Student C also discusses problem solving as an ability which might be useful after the present course:

> C: I don't know... I guess it's actually to solve a certain type of problem, it's rather like the math courses. Then learning different methods to solve them in different ways. Much like that, if you look back at the course it's not much actually but very, very fundamental. So to... get an overview and a basic idea of what it's about and that you can read on your own whenever you need.

3.2.5 *Learning is a skill that can be used outside the programming course*

The last category in Table 1 is *Learning to program is experienced as a skill to use outside the programming course.* This understanding presupposes the understandings in the previous categories. The ability to know and use a programming language, as expressed in categories one, two and four, are clearly expressed, but no longer the focus. The focus has moved outside the course and course context. The purpose of learning to program is not vaguely expressed as in category three. The students can clearly discuss why they want to learn to program and how they will use this knowledge after the course. Whilst this understanding is expressed in different ways by different students, what is common in the students' expressions is that the knowledge acquired is seen as something the student believe will prove useful later on, in further studies or in working life. Programming is experienced as a tool that will be beneficial for the student even after the course.

Student C focuses on the use of Java knowledge when learning other programming languages. Student C answers the question what he/she thinks the course is about:

> C: [...] But it feels as if you get a better grip on most languages, if you want to study C it will be easier after this course.

Student E also emphasizes the importance of independence. Knowledge is clearly described as a tool for his/her

own success, to be used to manage the working life better. Student E answers the question what he/she thinks the course is about:

> E: [...] I guess, it's ... learning to think like a programmer

Later in the interview:

> I: What's the point of learning to program [...]?
> E: Yes but it's that the more you know about computers the less dependent on others you'll be, sort of.
> I: I see.
> E: I don't know, if you work somewhere later and have some insight into things, then I think it'll open a window so that you know what it's about at least even if you don't, I mean, it's the pros that will deal with the real things.

Students who express an understanding belonging to category five, have managed to place the course and the course context in their own world and thinking about their future. Learning to program is experienced as meaningful for themselves, even though the reasons for this vary.

4. DISCUSSION

Table 1 presents the results of the study as qualitatively different ways of understanding of what it means to learn to program. We argue that it is crucial that the students reach category four, *Learning is a way of thinking, which enables problem solving, and which is experienced as a "method" of thinking*. In the understanding described in category two, the students have noticed that a special way of thinking is required, but not what that is. In category four, on the contrary, the students have realized that it has to do with problem solving and a systematic way of thinking. The interview excerpts indicate clearly that students who express an understanding corresponding to category two feel confused about programming.

Before we continue to discuss our results we want to point at the relation between the categories of understanding identified by us, and the discussion by Hazzan concerning 'process-object duality' [4]. This duality goes back to work by Piaget, and was developed in mathematics education to discuss the idea of reducing abstraction. Hazzan discuss this in terms of a passage from the 'process conception' to the 'object conception' [1]:

> *Process* conception implies that one regards a mathematical concept "as a potential rather than an actual entity, which comes into existence upon request in a sequence of actions." (Sfard, 1991, p. 4). When one conceives of a mathematical notation as an *object*, this notation is captured as one "solid" entity. Thus, it is possible to examine it from various points of view, to analyze its properties and its relationships to other mathematical notations and to apply operations on it. [4, p. 107 - 108] [9]

[1]When refering to our results, we will use the term 'category', while when refering to Hazzan's research, we use the term 'conception' to be consistent with her original terminology.

She concludes that according to these theories, "when a mathematical concept is learned, its conception as a process precedes - and is less abstract than - its conception as an object". It is thus a natural process when learning abstract concepts to start at the 'process conception'. The learning, in terms of process-object duality assumes however a passage from 'process conception' to 'object conception'. This is the desirable development also when learning computer science, including object-oriented programming.

Hazzan speaks of 'canonical procedures'. These are ways for the students to reduce abstraction level when dealing with concepts in different subjects. She writes:

> A *canonical* procedure is a procedure that is more or less automatically triggered by a given problem. This can happen either because the procedure is naturally suggested by the nature of the problem, or because prior training has firmly linked this kind of problem with this procedure. The availability of a canonical procedure enables students to obtain a solution without worrying too much about the mathematical properties of the concepts involved. It seems that this technical work gives students the assurance of following a well-known, step-by-step procedure, where each step has a clear outcome. In contrast, relying on abstract reasoning, for example by exploring properties of concepts or by relying on theorems, may be shaky mental approach for the students. Using the process-object duality terminology we may say that solving a problem by relying on a canonical procedure is an expression of process conception of the concepts under discussion; solving a problem by analyzing the essence and properties of concepts is an expression of object conception of the concepts under discussion. [4, p. 108]

The present study has its focus on students' understanding of what it means to learn to program, and more precisely to learn object-oriented programming. The process-object duality is of immediate interest in a course where we introduce abstract concepts like *object* and *class* early in the teaching, and where the understanding of these and other object-oriented concepts are fundamental for the rest of the course and for the ability to learn to program. Programming is a skill, but requires also a deep understanding of abstract concepts. "[A]nalysing the essence and properties of central concepts" in object-oriented programming is very much in line with the analysis and design phase in a programming problem. Analysis and design are abstract skills that belong to an 'object conception' that requires a good understanding of central concepts.

In object-oriented programming as in mathematics there are standard solutions to certain type of problems, 'canonical procedures' to learn and discover. They are used by experienced programmers, and necessary for the simplification and speed up of the work. It is thus desirable to help the students to discern such procedures. In this discussion we want to compare the students' discussion on 'programming thinking' when learning object-oriented programming, with a discussion on 'canonical procedures'. Many students mentioned 'programming thinking' as something specific, different from other subjects they had studied. Student D is an

example of this. He/she compares programming with other subjects studied.

> D: [...] I guess, it's just a rather different way of thinking.
> I: Now that you have taken this programming course could you put your finger on something you think is different than than chemistry... or you must have taken math too I suppose.
> D: Sure, I've taken many math courses but math is kind of logical and you understand it but this is... no I don't know (laughter). No but I kind of think it's easier to study math. Then you often have something creative to base it on, or you don't, but you learn more methods and kind of, there is some theory behind. Here you feel as if you only learn a lot of examples. You know, we've gotten so many examples of everything, in some way it feels as if you don't understand the base from the beginning [...]

Compare this when student C discuss what it means to learn in the present course:

> C: I don't know... I guess it's actually to solve a certain type of problem, it's rather like the math courses. Then learning different methods to solve them in different ways. Much like that, if you look back at the course it's not much actually but very, very fundamental.

Student C, who expresses an understanding belonging to category four has on the other hand obviously discerned 'canonical procedures', and has less problems in his/her learning. Student C has obviously reached the level of 'process conception', and seems to have reached further in his/her understanding.

Student D on the other hand has not even reached the level of 'process conception'. He/she explicitly finds it simpler to study mathematics because there they learn methods to use, which he/she obviously not has been given, or discerned in programming. Student D furthermore discusses how troublesome it is to know how to study programming. "But I still think the course, it's difficult to for a novice to get a good grip on how to study". This points to that student D is looking for 'canonical procedure' as a study technique, but has not found such to the extend he/she asks for. Student D also explicitly express that he/she founds it problematic to understand concepts within the subject and connects this to the ability to program "Yes, I think it has been difficult with concepts like that, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult, but this very thought behind, it feels as some people just understand programming".

In the understanding described in category four the students have realized that is has to do with problem solving and a systematic way of thinking. Using Hazzan's terminology, category four corresponds to 'canonical procedures', important in the learning process to reach the desired 'object conception'. It is not until category four that the students express an understanding of programming in terms of methods to use. This is therefor an important stage to reach. From the educators perspective, it is important to support students who have problem to reach 'object conception', to discern this understanding.

In our study some students do not even reach the level of 'canonical procedures'. Altough with different starting points Hazzan's research and the results from our study point to the same problem, but our study indicates that in learning object-oriented programming there are students who do not even reach a level of 'process conception'. Our main question is:

- **How can we help students to reach a level of 'object conception' in object-oriented programming?**

The present study indicates however that an earlier question educators need to ask is:

- **How can we help beginning programming students to discern 'canonical procedure' when learning object-oriented programming?**

Both Hazzan's results and ours emphasize that students might need 'canonical procedures' as a heave to reach the higher level of abstraction, the 'object conception'.

5. CONCLUSIONS

A study has been performed on beginning students' understanding of what it means to learn to program. The students were interviewed when they had just finished their first programming course. The phenomenographic analysis of the data identified what beginning students called 'programming thinking'. This is described e.g. as understanding of concepts, but also as methods and study techniques necessary when learning to program. The study indicates that some students have problems to discern the study techniques required when learning object-oriented programming, and this is also by the students connected to the learning of concepts. Some students commented that it is harder to discern such techniques in programming than in subjects like mathematics and chemistry.

By comparing our results with research on 'process-object duality', developed in mathematics education, we found that it is of great importance that students reach an understanding expressed as *learning to program is a way of thinking, which enables problem solving, and which is experienced as a "method" of thinking.* This corresponds to 'procedure conception', necessary for students to discern. Such a conception scaffolds for the more abstract level of understanding, the 'object conception'. This level of abstraction is described as "solving a problem by analyzing the essence and properties of concepts" and might be needed when central concepts in object-oriented programming are used in e.g. analysis and design. Both our results and Hazzan's point in the same direction, although from different starting points. This strengthen the discussion that students may need 'canonical procedures' for the passage to 'object conception'. Our study goes further in that it puts searchlight on the problems some students have to discern such 'canonical procedures' when learning object-oriented programming. As educators we need to become aware of the problems indicated, and facilitate for students to discern 'canonical procedures' in object-oriented programming. These can act as heaves to a more abstract understanding of central concepts within object-oriented programming.

6. REFERENCES

[1] A. Berglund. *Learning computer systems in a disributed project course. The what, why, how and where*. PhD thesis, Uppsala University, Department of Information Technology, 2005.

[2] S. A. Booth. *Learning to Program. A phenomenographic perspective*. Number 89 in Göteborg Studies in Educational Science. Acta Universitatis Gothoburgensis, Göteborg, Sweden, 1992.

[3] C. Bruce, C. McMahon, L. Buckingham, J. Hynd, M. Roggenkamp, and I. Stoodly. 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, 2004.

[4] O. Hazzan. How students attempt to reduce abstraction in the learning of computer science. *Computer Science Education*, 13(2):95–122, 2003.

[5] M. Kölling. The problem of teaching object-oriented programming, part i: Languages. *JOURNAL OF OBJECT-ORIENTED PROGRAMMING*, January 1999.

[6] S. Kvale. *InterViews: An introduction to qualitative research interviewing*. Sage, 1996.

[7] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Ass., Mahwah, NJ, 1997.

[8] E. Roberts. The dream of a common language: The search for simplicity and stability in computer science education. In *Proceedings of the thirty-fifth SIGCSE technical symposium on Computer science education*, 2004.

[9] A. Sfard. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 22:1–36, 1991.

Paper VI

Successful Students' Strategies for Getting Unstuck

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT USA
robert@cse.uconn.edu

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Kate Sanders
Department of Math and Computer Science
Rhode Island College
Providence, RI USA
ksanders@ric.edu

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA USA
zander@u.washington.edu

ABSTRACT

Students often "get stuck" when trying to learn new computing concepts and skills. In this paper, we present and categorize strategies that successful students found helpful in getting unstuck. We found that the students reported using a broad range of strategies, and that these strategies fall into a number of recognizably different categories.

Categories and Subject Descriptors

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

General Terms

Measurement, Experimentation

Keywords

Learning strategies, stuck places, Threshold Concepts

1. INTRODUCTION

Learning does not always occur at a constant rate of increasing knowledge and skills. While learning new concepts and skills, students sometimes encounter epistemological obstacles [11]—that is, they get stuck and are unable to make progress toward learning and understanding.

In this paper we look at strategies that successful computing students reported using to become unstuck and make progress in learning these concepts and skills. These students are successful in two ways: one, they successfully learned particular computing concepts after being stuck, and two, they have been successful in their educational programs: the students we interviewed were within a year of graduation, all due to finish their degree programs by the end of 2006.

The goal of this investigation is twofold. First, we would like to identify strategies that students use successfully in their computing studies. Second, we would like to categorize these strategies in ways that make them useful for future students and instructors. We have approached this investigation from the student perspective, focusing on what students report about their learning.

Section 2 describes the techniques we have used to gather information about getting unstuck from the student's perspective, the student strategies we identified, and a hierarchy into which those strategies can be organized. In Section 3 we provide examples from interviews that illustrate the range and depth of the students' use of learning strategies. In Section 4, we compare our results to those reported in the general and computing education literature. In Section 5, we give some overall impressions of the way that students used these strategies, and how we might use this information in making instruction more effective. Finally, in Section 6, we present our conclusions and discuss the future directions of this research effort.

2. DATA GATHERING AND ANALYSIS

The data used here were gathered using semi-structured interviews, as part of a larger investigation into *Threshold Concepts* in computing [2, 4]. Fourteen students (total) were interviewed at six institutions in Sweden, the United Kingdom, and the United States. For analysis, the student interviews were transcribed verbatim; where necessary, they were translated into English by the interviewer.

Some of the interview questions dealt explicitly with the idea of being stuck and becoming unstuck—these were used to identify possible threshold concepts to pursue in depth. The parts of the script dealing with these topics are given in Figure 1 (for a more complete script, see [2]). The students provided a surprisingly rich list of strategies they used to get unstuck, along with advice for other students in similar situations.

ITiCSE'07, June 23–27, 2007, Dundee, Scotland, United Kingdom.

1. Could you tell me about something where you were stuck at first but then became clearer? *(Subject answers <X>.)*

   The rest of this session will now focus on <X>.

2. Can I start by asking you to tell me your understanding of <X>?

4. Tell me your thoughts, your reactions, before, during and after the process of dealing with <X>.

5. Can you tell me what helped you understand <X>?

7. Based on your experience, what advice would you give to help other students who might be struggling with <X>?

13. To finish the interview, can you tell me whether there are any other things where you were stuck at first but then became clearer?

Figure 1: Interview script excerpt: parts concerning stuck places, getting unstuck, and strategies.

Impressed by the students' responses, we examined this portion of our data from a new angle. After extracting the quotes relevant to this topic, we proceeded inductively, identifying and naming 35 distinct strategies, grouping those into 12 more abstract categories, and finally grouping those 12 into four super-categories, all of which in turn are examples of *Get unstuck/learn*. We worked individually, discussed with the group until we reached agreement, and referred back to the original quotes as needed.

The 35 basic strategies, the 12 categories, and the 4 super-categories are shown in Table 1; the top levels of the hierarchy (from *Get unstuck/learn* on down) are shown in tree form in Figure 2.

3. WHAT THE STUDENTS SAID

Space does not permit inclusion of all the interesting quotes, or even quotes illustrating all 35 of the basic strategies. In this section, we give some of the most interesting quotations, illustrating each of the four super-categories.

3.1 Inputs/interaction

Many students talked about getting help from elsewhere. Not surprisingly, subjects read, looked information up on the Internet, and used tools. *Subject12* says

> *... instead of basically doing it I would sit there and read trying to figure out how to do it ...*

Subject9 demonstrates a common tendency to rely on tools:

> *... helped in Java doc and API on the Internet.*

Subjects frequently learned from other people. *Subject11* discusses getting the information needed to figure out problems from a variety of sources:

> *Like either be it peer or, you know, another faculty member that, you know, understands the problem.*

It was suggested by *Subject7* for instructors to give step-by-step instructions that students can follow until they are comfortable with the material:

Table 1: Identified strategies and their abstractions

Strategy	Abstract strategies	
Discuss	Learn from other people	Inputs/interactions
Learn from peers		
Listen to professor		
Get help (from a person)		
Read	Learn from tools or written materials	
Use a tool		
Get and follow step-by-step instructions	Get and follow step-by-step instructions	
Remember things		"Use the Force"
Be persistent/don't stop		
Avoid the problem / work around		
Walk away and come back later		
Reflect/sit and think		
Write programs	Gain experience	Concrete/do stuff
Learn by trial and error		
Learn from your mistakes		
Practice/drill		
Visualize/see a diagram	Visualize	
Draw diagrams/pictures		
Connect diagrams with code		
Use examples (in general)	Learn from examples	
Use varied examples		
Use sequence of increasingly complex examples		
Trace	Trace	
Break into parts	Divide and conquer	Abstract/understand stuff
Use incremental development		
Model real world	Relate to real world	
Use analogy to real world		
See context/reason/ use for something	Look for the bird's eye view	
See the larger picture		
See patterns		
See a large system		
Transfer from language to language	Make transfers/ connections	
Connect to mathematical formalism		
Relate to something already learned		
Relate different levels of abstraction		

> *So you could as a teaching tool, you could say, now we're going to do this and these are the steps to putting these things together, just like you did with recursion. Step one, write out what you are going to do. Step two, write out what the defining check, whatever - those things. In this program we'll do this, this, and this. Just trust me. It's going to work.*

3.2 Concrete/do stuff

Often a single quote showed multiple strategies. Practicing and learning from examples was a common combination with getting help from others. *Subject3* discusses the instructor, learning from examples, and practice:

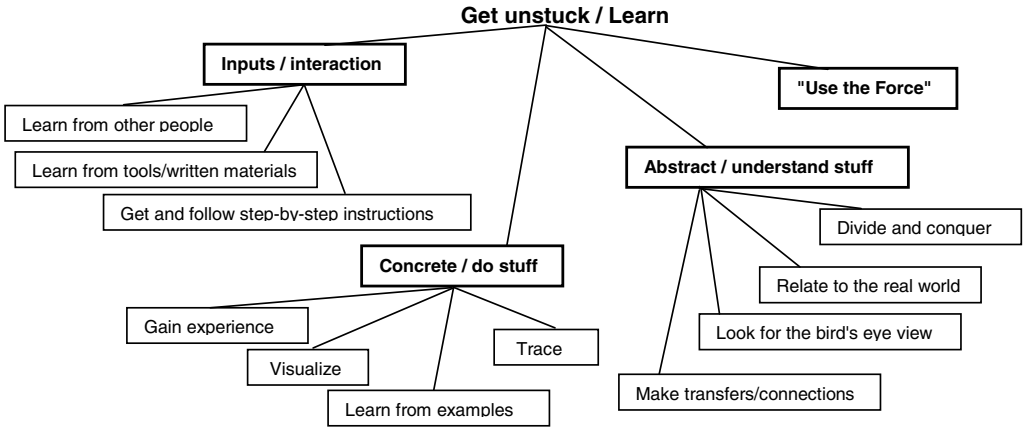

Figure 2: Hierarchy of abstract strategies from Table 1.

> *Go to your instructor, get plenty of examples. And do a lot of, again, do a lot of little programs. And don't be afraid, jump right in.*

Subject7 learns from discussion with peers and lots of visualization:

> *I can't tell you how many dry erase markers we've gone through over there covering entire boards with drawings and drawings and drawings and arrows, this constant sort of thing in getting procedures and taking the procedure and drawing the picture from it.*

Recommending lots of practice comes up in relation to many different kinds of problems. Here, *Subject6* recommends practice when discussing object-oriented programming and concurrency:

> *I would say, deviate from the assignments on your own time and write programs that you think are completely useless and stupid. You think of these programs, no one's going to use them. ... Just do it anyway because you'll understand. You'll run into problems and you'll find the solutions to that problem. [...] And then, when the school project does come, you'll have had the experience from what you've done on your own. But I think it's important that you don't just do the schools. You've got to do it on your own.*

3.3 Abstract/understand stuff

In contrast, some subjects discussed learning and getting unstuck at a higher level. Many quotes relate concepts to real world examples, relate pointers to a television remote control or a leash, relate objects to different rooms in a house. And, while *Subject1* talks about examples, it is the idea of putting it in context that stands out:

> *But to see all of those different [examples]—that same idea in all those different contexts and to figure that out on my own really just taught me like critic-like thinking skills.*

Several subjects talk about breaking problems down into smaller parts, and *Subject10* discusses the upward view, the larger picture in object-oriented design:

> *Once you do it enough you stop thinking about that and you think about it in a bigger view if that makes sense.*

Furthermore, *Subject4* addresses the importance of seeing patterns:

> *I think if a person can see the pattern, I think I'm no different from anyone else. If I can see the pattern, I can generally, I can take a technique and I can go home and figure it out if there's a pattern to it. I understand the pattern, why the pattern fits, and I can see how to figure out the exceptions to those patterns.*

And, *Subject1* demonstrates the importance of making connections and formalism:

> *And it wasn't until I took really functional programming after discrete math that I realized solutions could be simpler if I used recursion, less lines or just easier to code or easier to reason about.*

3.4 "Use the Force"

A number of strategies involve the students using their willpower or character: telling themselves to remember things, to be more persistent, or to sit and think. For example, the following quote from *Subject4* illustrates the value of remembering things:

> *And then you just, you know, you get that little nugget in your head and you carry that on to the next time. So, the next time you have something*

with a pointer that isn't working, you go, Okay, I need to do this. And if it still isn't working, then it's something else.

Subject11 talks about the value of being persistent:

And just by just staring at it and continually like trying to grasp it, I eventually got like a small piece of it and understood it.

These are generalized strategies that can be applied on top of other specific strategies. The name reflects the aspects pertaining to personal characteristics, an allusion to the movie *Star Wars*, where the main character is admonished to "Use the Force"—to trust his intuition or inner strength.

4. RELATED WORK

Perkins [12] discusses three different sorts of learners—*active, social,* and *creative*—and how they might respond to different forms of constructivist learning. These groups match up fairly well with our abstract strategies: *Concrete/do stuff* for the active learners, *Inputs/interaction* for the social learners, and *Abstract/understand stuff* for the creative learners. Work on "learning styles"[5] indicates that different students may prefer different strategies; for example, visual learners are more likely to draw diagrams, and verbal learners, to read. In any case, our data suggest that successful students often apply multiple strategies to learning a given concept. McKeachie et al. [10] found that the teaching of learning strategies in a course provided benefits to students in their other courses.

Biggs [1] stresses the importance of student activities (as opposed to instructor activities) in learning. Ramsden ([14], p.155) agrees: "Passivity and dependence on the teacher [...] provide an excellent basis for surface approaches" and "deep approaches are associated with activity and responsibility in learning." The strategies seen used by these students are nearly all centered on student activities: even the *Learn from other people* strategies tend to be driven by student actions, such as seeking out peers or others to learn from. Trigwell et al. [16] discuss the role of "students' intentions associated with their strategies"; the students interviewed here seem to have been motivated to learn, at least in their retrospective explanations.

A number of studies consider the relative value of deep vs. surface approaches to learning, see e.g. [9], generally with the assumption that deep is inherently better. This view is not exactly supported by work of Hughes and Peiris [7], who investigated students' approaches to learning programming in terms of deep, surface or strategic approaches– the strategic approach being to "plan their work according to an awareness of tutor's expectations." The results showed that course performance had a strong negative correlation with taking a surface approach, no strong correlation with taking a deep approach, but a strong positive correlation with using a strategic approach. Further, Kember and Gow [8] found cultural differences in the relative effectiveness of surface and deep approaches. We saw a range of successful strategies from surface *(Practice/drill)* to deep *(Connect to mathematical formalism)*; we did not see Hughes and Peiris's strategic approaches, but we did not discuss general motivation in our interviews.

Debugging is one area of getting unstuck which has been studied in depth. In particular, Vessey [17] identified a hierarchy of goals used by programmers while debugging. Novices stated more hypotheses than experts and tended to stick with their hypotheses so that they failed to understand the program structure.

In addition to work on learning strategies, there has been a good deal of work on student strategies used in programming, program comprehension, and problem solving.

Robins et al. [15] identify lack of programming strategies as a cause for problems for novice programmers. Davies [3] reviewed "studies that have addressed the strategic aspects of programming skill", and suggests that "the strategic elements of programming skill may, in some case, be of greater significance than the knowledge-based components."

Fitzgerald et al. [6] identified 19 strategies used by novice programming students when solving examination questions that involved reading and understanding code. They found that success was determined both by the strategies chosen and how well they were employed.

There is quite a large literature on problem-solving strategies, most notably Pólya [13]. Wankat and Oreovicz ([18], chapter 5) summarize a large numbers of studies comparing novice and expert problem solvers. Although problem solving and concept learning are quite different, we see some similarities. The stated expert responses to being stuck are given as "Use heuristics", "Persevere", and "Brainstorm." While the first of these was not seen in our data (except perhaps *Use analogy to real world*), the others are seen as *Be persistent/don't stop* and *Discuss*.

5. DISCUSSION

None of the strategies mentioned by any student was surprising in itself. The group of interviewed students, however, supplied a surprisingly long—and thoughtful—list of suggestions. Much of what they suggested was social: several of the students pointed out that it was with the help of their friends, or by asking their instructor or another faculty member, that they actually learned. They mentioned how important it is to relate new knowledge to something you already know. They said that they like step-by-step instructions when learning something new. They stressed the value of persistence and practice, even inventing tasks above and beyond the assigned work.

Some immediate implications for teaching are that we can give the students what they (or some of them) find helpful: the opportunity to discuss their work with their peers, good examples that relate to what they already know, step-by-step instructions, optional extra tasks, etc.

But experienced instructors already do these things. This study provides a deeper understanding of student experience, and a list of possible ways to get unstuck. Just as we might teach debugging—what to do when your program doesn't work—we can also explicitly discuss how to debug your own learning.

6. CONCLUSIONS AND FUTURE WORK

We have investigated graduating students' successful strategies for getting unstuck when learning new concepts. All but one of our students recalled getting stuck at one point or another—it is unlikely that a student makes it through a computing degree program without this experience! But

the successful students, it seems, have strategies for learning. Although they get stuck, they find a way to get unstuck.

Altogether we found 35 strategies which we have categorized into groups. The result shows that all students used several strategies, and that the strategies are surprisingly diverse. The importance of social interaction, and the active responsibility taken by the students are striking. This is consistent with previous research emphasizing the importance of students taking responsibility for their own learning. Many strategies discussed by the students can be used either with a surface or a deep approach. The data show that our successful students have used the strategies with a deep approach, actively striving for learning.

It may seem obvious but we think it is worth pointing out that the students experienced and overcame their problems in different ways. It is important for us as teachers to not only acknowledge this but also plan our courses with this in mind. We do not think there is a "one-size-fits-all" recipe for how to learn computing concepts; instead teachers should be aware of strategies students use and regard as successful, and be prepared both to encourage the students to use the strategies they prefer, and to help them to learn new ones.

The results so far suggest a number of areas for future study:

- Learning strategies that students use when they are not stuck. Are other successful strategies used in more "routine" situations that were not considered in this study?
- Learning strategies used by *unsuccessful* students—either strategies used unsuccessfully, or by students who fail to complete their degrees. Do these students use different strategies? Do they attempt to use the same strategies without success?
- The use of strategies by students at various points in their degree programs. Do the use of strategies develop as students gain experience, or do they gain experience using strategies they always knew, or a mix?
- Applicability of strategies. Are there correlations between the strategies used and either particular concepts being learned, or the preferred learning style of the student?

We think that a better understanding of the learning strategies that students use (and could be taught to use) could lead to improved learning and teaching. We (the authors) have found that what we know so far has already affected how *we* teach in a positive way.

Acknowledgments

The authors would like to thank Mark Ratcliffe and Jonas Boustedt, who participated in the design, data collection, and initial analysis of the Threshold Concept interviews. Thanks also the Department of Information Technology at Uppsala University for providing us with workspace and facilities in Uppsala, and to Sally Fincher, Josh Tenenberg, and the National Science Foundation (through grant DUE-0243242) who provided workspace at the SIGCSE conference in Houston. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Uppsala University.

7. REFERENCES

[1] J. Biggs. *Teaching for Quality Learning in University*. Society for Research in Higher Education and Open University Press, Buckingham, 1999.

[2] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Threshold concepts in computer science: do they exist and are they useful? In *SIGCSE-2007*, pages 504–508, Covington, KY, March 2007.

[3] S. P. Davies. Models and theories of programming strategy. *Int. J. of Man-Machine Studies*, 39(2):237–267, 1993.

[4] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Putting threshold concepts into context in computer science education. In *ITiCSE-06*, pages 103–107, Bologna, Italy, June 2006.

[5] R. Felder. Reaching the second tier: Learning and teaching styles in college science education. *J. College Science Teaching*, 23(5):286–290, 1993.

[6] S. Fitzgerald, B. Simon, and L. Thomas. Strategies that students use to trace code: an analysis based in grounded theory. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 69–80, Seattle, WA, USA, 2005.

[7] J. Hughes and D. R. Peiris. ASSISTing CS1 students to learn: learning approaches and object-oriented programming. In *ITiCSE-06*, pages 275–279, Bologna, Italy, June 2006.

[8] D. Kember and L. Gow. A model of student approaches to learning encompassing ways to influence and change approaches. *Instructional Science*, 18:263–288, 1989.

[9] F. Marton, D. Hounsell, and N. Entwistle. *The Experience of Learning*. Scottish Academic Press, Edinburgh, 1984.

[10] W. J. McKeachie, P. R. Pintrich, and Y.-G. Lin. Teaching learning strategies. *Educational Psychologist*, 20(3):153–160, 1985.

[11] J. H. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49:373–388, 2005.

[12] D. Perkins. The many faces of constructivism. *Educational Leadership*, 57(3):6–11, 1999.

[13] G. Pólya. *How to Solve It*. Princeton University Press, Princeton, NJ, 2nd edition, 1957.

[14] P. Ramsden. *Learning to Teach in Higher Education*. Routledge, London, 1992.

[15] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137 – 172, 2003.

[16] K. Trigwell, M. Prosser, and P. Taylor. Qualitative differences in approaches to teaching first year university science. *Higher Education*, 27(1):75–84, 1994.

[17] I. Vessey. Expertise in debugging computer programs: Situation-based versus model-based problem solving. *Int. J. of Man-Machine Studies*, 23:459–494, 1985.

[18] P. C. Wankat and F. S. Oreovicz. *Teaching Engineering*. McGraw-Hill, New York, 1993.

Paper VII

Computer Science Education
Vol. 16, No. 3, September 2006, pp. 197–209

Routledge
Taylor & Francis Group

Categorizing Student Software Designs: Methods, results, and implications

Anna Eckerdal[a]*, Robert McCartney[b], Jan Erik Moström[c], Mark Ratcliffe[d] and Carol Zander[e]
[a]*Uppsala University;* [b]*University of Connecticut;* [c]*Umeå University;* [d]*University of Wales Aberystwyth;* [e]*University of Washington Bothell*

This paper examines the problem of studying and comparing student software designs. We propose semantic categorization as a way to organize widely varying data items. We describe how this was used to organize a particular multi-national, multi-institutional dataset, and present the results of this analysis: most students are unable to effectively design software. We examine how these designs vary with different academic and demographic factors, and discuss the implications of this work on both education and education research.

1. Introduction

A fundamental goal of computer science programs is that their graduates be able to design software systems. This suggests that it is important to assess whether this goal is being met. However, no method is agreed upon to do this.

A fairly direct approach would be to analyze student-produced designs for a common set of tasks. If students are allowed to design as they wish, however, the data produced will be very rich and varied, even in controlled settings. Although this is positive in that it can illustrate the range of approaches that students take, the complexity and variability of the designs makes them difficult to analyze and compare.

In this study, we worked with written designs produced by near-graduating students under stringent interview conditions. These designs were collected as part of a larger study, the ''Scaffolding'' experiment; a multi-national, multi-institutional project looking at the approach students take to software design.

The Scaffolding study (Blaha, Monge, Sanders, Simon, & VanDeGrift, 2005; Chen, Cooper, McCartney, & Schwartzman, 2005; Tenenberg et al., 2005) looked at two different groups of students, novices and near-graduates, as well as their educators. Each subject performed two tasks, a *design* task, in which they designed a software system from its description, see Figure 1, and a *design criteria prioritization* task, in which they ranked software design criteria by importance under different

*Corresponding author. E-mail: Anna.Eckerdal@it.uu.se

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/06/030197-13 © 2006 Taylor & Francis
DOI: 10.1080/08993400600912376

Design Brief

Getting People to Sleep

In some circles sleep deprivation has become a status symbol. Statements like “I pulled another all-nighter” and “I’ve slept only three hours in the last two days” are shared with pride, as listeners nod in admiration. Although celebrating self-deprivation has historical roots and is not likely to go away soon, it’s troubling when an educated culture rewards people for hurting themselves, and that includes missing sleep.

As Stanford sleep experts have stated, sleep deprivation is one of the leading health problems in the modern world. People with high levels of sleep debt get sick more often, have more difficulties in personal relationships, and are less productive and creative. The negative effects of sleep debt go on and on. In short, when you have too much sleep debt, you simply can’t enjoy life fully.

Your brief is **to design a "super alarm clock" for University students** to help them to manage their own sleep patterns, and also to provide data to support a research project into the extent of the problem in this community. You may assume that, for the prototype, each student will have a Pocket PC (or similar device) which is permanently connected to a network.

Your system will need to:

- Allow a student to set an alarm to wake themselves up.
- Allow a student to set an alarm to remind themselves to go to sleep.
- Record when a student tells the system that they are about to go to sleep.
- Record when a student tells the system that they have woken up, and whether it is due to an alarm or not (within 2 minutes of an alarm going off).
- Make recommendations as to when a student needs to go to sleep. This should include "yellow alerts" when the student will need sleep soon, and "red alerts" when they need to sleep now.
- Store the collected data in a server or database for later analysis by researchers. The server/database system (which will also trigger the yellow/red alerts) will be designed and implemented by another team. You should, however, indicate in your design the behaviour you expect from the back-end system.
- Report students who are becoming dangerously sleep-deprived to someone who cares about them (their mother?). This is indicated by a student being given three “red alerts" in a row.
- Provide reports to a student showing their sleep patterns over time, allowing them to see how often they have ignored alarms, and to identify clusters of dangerous, or beneficial, sleep behaviour.

In doing this you should (1) produce an initial solution that someone (not necessarily you) could work from (2) divide your solution into not less than two and not more than ten parts, giving each a name and adding a short description of what it is and what it does – in short, why it is a part. If important to your design, you may indicate an order to the parts, or add some additional detail as to how the parts fit together.

Figure 1. The design brief that was given to the subjects

design scenarios. These tasks were performed one-on-one with a researcher who took notes, answered questions, and made an audio recording of the session. In addition, academic and demographic information was collected for each subject: age, gender,

grades, number of CS courses taken, knowledge of different programming languages, and so forth. The overall focus of the study was developmental—so its questions were largely comparative, as in "How do beginning students, finishing students, and educators differ in the way they design software?"

By contrast, we are specifically interested in students at the end of their academic training, when they are presumably prepared to work professionally. The overall question is: Can students near graduation design software systems? In order to address this and other related questions, it is necessary to characterize the designs for evaluation and comparison. We did this by grouping the designs into categories that were meaningful for our purposes, and then correlating the categories of the designs with academic and demographic information about the students.

The structure of this paper is as follows. First, we describe the processes by which we categorized and analyzed the designs, and present student design results. Then we discuss these results in light of other design research. Finally, we consider the implications of this work for educators and researchers, and provide suggestions for future study.

2. Categorization Methods

The goal of the present study was to examine students' abilities to design software, using their written designs as the primary data. To organize and simplify these data for analysis, we categorized them into groups of similar designs. We chose a data-driven approach for this categorization, with the intention that the categories reflect similarities that we observed in the data. Moreover, we intended that the observed similarities and differences be meaningful relative to the design task.

2.1. Categorization and Classification

We categorized, as opposed to classified, these designs. As in Jacob (2004) we define *categorization* as an assignment of items to categories based on semantic similarity in context (design in our case). Category members can be more or less typical, so categories can meaningfully be based on prototypes and have fuzzy boundaries.

We grouped designs based on their semantics, that is *what* they communicate rather than *how*, and how well they met the stated requirement in the design brief that the design be something that "someone (not necessarily you) could work from". Based on this approach, we developed six categories of designs, shown in Figure 2, ordered relative to the degree to which the stated requirement was met.

These category descriptions have a distinctive characteristic: they include a description of category members in general, and refer to a typical example, or *prototype*[1]. The descriptions include qualitative terms without clear boundaries. For example, "add a small amount" in *Skumtomte*[2] and "include some significant work" in *First step* both refer to amounts of added information; the prototypes provide examples of these amounts. Choosing the closest prototype allows artifacts to be placed in categories even though the boundaries are inexact, but suggests that it may be difficult to precisely categorize some of the artifacts.

Nothing This category has designs with little or no intelligible content. These tend to be very short, typically a single unlabelled diagram. There are very few of these.

Restatement These are designs that merely restate requirements from the task description (Figure 1). A typical example is a list of functions that correspond to the bulleted items in that description. These have no design content that was not stated in the description.

Skumtomte These are designs that add a small amount to restating the task. Some subjects added a small amount of information in text, or remedial GUI (Graphical User Interface) drawing, or some unimportant implementation details. There is no overall system view, nor is there any significant work on any of the modules. A design often has a list restating the bullet items, plus one or two details.

First step Designs in this category include some significant work beyond the description: either a partial overview of the system (identifying the parts, but generally not identifying their relationship) or the design of one of the system's components, such as the GUI or the interface to the database. There are two prototypical members. One has a partial overview expressed as a UML (Unified Modeling Language)-like diagram (or in text) that identifies the system modules but not their interactions; the other gives a fairly detailed description of the GUI, but provides no system overview.

Partial design A partial design provides an understandable description of each part and an overview of the system that illustrates the relationships between the parts. The parts descriptions may be incomplete, and the communications between the parts is not completely described. A design describes the parts, and has an architecture diagram with links between communicating parts, but no description of what is being communicated.

Complete These designs show a well-developed solution, including an understandable overview, part descriptions that include responsibilities, and explicit communication between the parts. A typical example uses multiple formal notations, e.g., UML, Use cases, CRC (Class, Responsibility, and Collaboration) cards, as well as text.

Figure 2. The six categories used for design artifacts

2.2. Developing the Categories and Tagging Designs

The processes of developing the categories and assigning the designs to categories were data-driven—both the category descriptions and the previous design assignments changed as the category assignment, or *tagging*, progressed.

The initial categorization was based on examination of 20 randomly chosen designs. After a number of attempts based on syntactic features, one researcher proposed a categorization based on semantic design content. This first attempt had five categories.

Based on these descriptions and prototypical designs, each researcher individually tagged a group of 70 designs. Considering all pairwise researcher comparisons, around 60% of the tags agreed. To reach agreement on the categories, four of the researchers met and considered the designs in detail. The categorization was revised

as the differences were resolved, so that the categorization would best reflect observed patterns in the data. For more details on the categorization process, see Eckerdal, McCartney, Moström, Ratcliffe, and Zander (2006b). The resulting distribution of designs in categories is shown in Figure 3.

In terms of analysis, categorizing the designs was quite time-consuming, even after the categories were defined, as it required extracting the meaning from the artifacts, many of which are poorly organized and nearly illegible. Gaining complete agreement between raters required extensive discussion for designs that did not closely match the prototypes or were difficult to read. Having placed designs in these categories, however, gave us useful information about the communicated design content, and allowed us to easily compare designs on that basis.

2.3. Syntax and Semantics

Extracting meaning from a design, which is necessary to categorize it, is difficult as it requires a global understanding of the artifact. Recognizing syntactic features, by contrast, does not require such deep understanding as they are visually apparent. We examined the relationships between our semantic categories and the recognizable syntactic features.

Before and during the categorization, we identified a number of candidate syntactic features that might be used to characterize designs. Listing these and collapsing similar features, we agreed to use the set in Table 1.

We then re-examined all of the designs relative to this feature set, determining for each design, whether each feature was present. (We did not attempt to count how

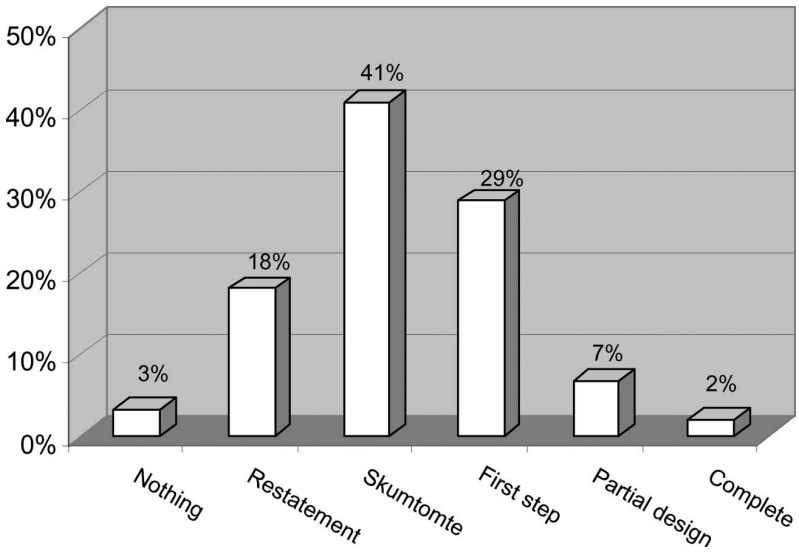

Figure 3. Frequencies of observations in each of the design classifications (based on 149 observations)

many of each feature were present.) We compared the number of different features present in each design for each of our semantic categories. We summarize this in Table 2. This table shows that the feature count increases as the designs become more complete. The designs in higher categories tend to be syntactically richer, in part because they tend to include some of the formal structures like UML and Use case diagrams. As the table further indicates, however, there is a large range of feature counts in each category.

If the number of features measures the syntactic richness of the design, the overall length of the design is a measure of the syntactic quantity of the design. As seen in Table 3, the higher category designs tend to be longer. As with the feature count,

Table 1. The syntactic features considered

Algorithm	A step-by-step description.
Block	Box with text in it, usually a single word.
Bulleted list	A bulleted, numbered, or labeled list of short items.
Class	A class, represented in code or in a diagram (*not counted separately if in UML diagram*).
Code	Code snippets, for example, assignment statements.
CRC	CRC cards explicitly represented.
Database	A detailed representation of the database.
Event-action	The design is described as ''when X happens the system should do Y'', more elaborate than single sentences.
Flowchart	A graphical flowchart.
Methods	A method (function) is described. *Not counted separately if included in a class description or in a UML diagram*.
Other diagram	Miscellaneous drawings (*not counted elsewhere*).
Overview diagram	A diagram showing an overview of the main design parts.
Running text	More than a couple of sentences of text.
Simple UI	A simple drawing of the user interface.
Text outline	A hierarchical outline of short items.
User Interface	An elaborate drawing of the user interface.
UML	A UML diagram.
Use case	A use case description.
User picture	The users of the system are explicitly drawn similar to stick figures. *Not counted if part of a Use case*.

Table 2. The average, minimum, and maximum number of features counted per design, by category

Number of features	Nothing	Restatement	Skumtomte	First step	Partial design	Complete design
Average	1.20	1.30	2.57	3.30	4.20	4.67
Minimum	1	1	1	1	2	4
Maximum	2	3	6	5	6	6

Table 3. Length of designs by category. Entries are in number of pages except for last row, where entries are percent of designs at least three pages long

Number of pages	Nothing	Restatement	Skumtomte	First step	Partial design	Complete design
average	1	1.9	2.9	3.7	5.2	5.0
minimum	1	1	1	1	2	2
maximum	1	4	9	9	10	9
≥ 3	0	18.5%	54.8%	66.7%	90%	66.7%

there is a great deal of variance, however, as illustrated by the minimum and maximum values for each category.

In summary, semantic content does correlate with syntactic richness (number of features) and syntactic content (as measured by length). However, these syntactic measures are quite variable within categories, so they make relatively poor predictors of category. If the meanings of the designs is what is important, a semantic categorization should be more valuable.

3. Student Design Results

The distribution of the designs among the categories is shown in Figure 3. Given that the categories can be naturally ordered relative to the communicated design content, we can describe the overall performance:

- 21% of the designs were simply restatements of the specification or less—no value added at all.
- 41% of the designs were *Skumtomte*: those that added an insignificant amount beyond the specification, and, in particular, did not produce any usable ''design content''.
- 29% of the designs were in the First step category, showing some progress toward a design—a partial overview, or significant progress toward the design of one part of the system.
- 9% produced *Partial* or *Complete* designs: those including an understandable system architecture/overview, with parts and their interactions explicitly stated. Of these, less than one third produced *Complete* designs, with explicit part responsibilities and inter-part communications.

All in all, a poor performance from students who are near graduation: over 20% produced nothing, and over 60% communicated no significant progress toward a design.

As part of the overall study, we collected academic and demographic background data on the students and made other observations during the design task including:

their age and gender; their academic background (grades in computer science courses, number of computer science courses taken, number of programming languages known, number of programming languages known well), and the time they spent on the design task. For example, Figure 4 plots the average number of CS courses taken versus the design categories produced, showing a positive correlation between the completeness of the design and the number of courses taken.

To measure the correlation between these factors and the categories, we assigned the numbers 0 (Nothing) to 5 (Complete) to the categories and calculated Pearson's r as a correlation measure. We used a t-test at $\alpha = .05$ to check whether that correlation was significant. We observed the following (values for r and the attained significance p at 147 degrees of freedom are in parentheses):

- The number of courses taken ($r = .362$, $p < 10^{-5}$), the time spent on the task ($r = .420$, $p < 10^{-7}$), and the number of programming languages known well ($r = .179$, $p = .0289$) were significantly (positively) correlated with the design categories.
- Grades in CS courses ($r = .040$, $p = .6307$), age of participant ($r = .097$, $p = .2394$), and the number of programming languages known ($r = -.030$, $p = .7153$) were not significantly correlated with the design categories.
- There seemed to be qualitative gender differences (females had relatively more designs in the top three groups, and fewer at the extremes), but the number of females was quite small (15%).

More details can be found in Eckerdal, McCartney, Moström, Ratcliffe, and Zander (2006a).

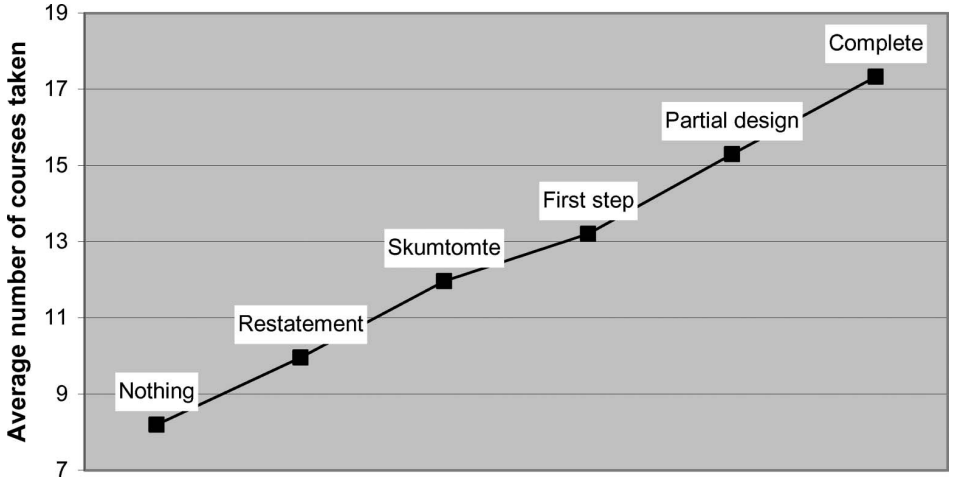

Figure 4. Average number of CS courses completed

4. Comparison with Related Work

As described above, we have not tried to put the designs into some pre-defined categories. Instead we let the data speak for itself. Comparing our results with previous work suggests that this has been a judicious approach.

While there has been much research over the years into student coding (Robins, Rountree, & Rountree, 2003), there is far less available on design. Results from both of these areas, however, are consistent with what we have seen here.

Of particular interest to student design is recent work by McCracken (2004) which focuses on the *learning* of design skills. He contrasts design and programming, and suggests techniques for studying design behavior: *in-situ* observation, Retrospective interviews, and Protocol analysis. The Scaffolding data collection was an example of *in-situ* observation in his terms. The big difference between his approaches and ours is that he concentrates on the *process* of designing as opposed to the *results*. We chose not to use the collected process data from the Scaffolding study: differences in experimental technique among the researchers make these data difficult to compare, and the extracting of the subject behaviors is too labor intensive for the size of the dataset.

DuBoulay comments on novice programmers' inability to grasp the whole program and the relation between the main parts: "This ability to see a program as a whole, understand its main parts and their relation is a skill that grows only gradually" (DuBoulay, 1986, p. 59). This is consistent with our conclusions that overview of the parts and relations between parts are important features found only in the more advanced designs.

In terms of techniques used, Atman, Chimka, Bursic, and Nachtmann (1999) provides a good contrast. In this study, Atman and her collaborators compared the design *process* between freshman and senior engineering students using verbal protocol analysis: students were observed and recorded while "thinking aloud" as they performed a design task. To capture and analyze the design process, they did the following:

- Transcribed the tapes.
- Segmented the transcripts (that is, separated the transcripts into units of one idea) with multiple raters, then resolved differences.
- Coded each segment with regards to four variables: in what stage of design did this happen (10 possibilities); what activity was being done (5 possibilities); what type of information was being addressed (>8 possibilities), and which object in the design was being considered (>8 possibilities). As above, done with multiple raters followed by difference resolution.

Given these coded segments, they examined the emphasis on each activity (total and relative time spent on different activities); the pattern of the effort—how the designer moved from one activity to the other; the amount and kinds of information gathered by the designer; and the number of alternative designs considered. They also rated the quality of the designs using multiple measures.

The obvious difference between this work and ours is the amount of effort required to do the analysis. They had extremely rich data: audio and videotaped design sessions up to three hours long, plus all of the produced design artifacts. They were able to compare the design processes as well as outcomes, and found interesting differences between their two participant groups. Getting the data into the form necessary to make process comparisons, however, was extremely expensive, which puts practical limits on the applicability of this approach. One other big difference, not surprising given their emphasis on process, is how they expressed the outcomes. Their quality measure was a complex formula based on many factors, but all of these factors were ultimately combined into a single number between 0 and 1—possibly providing less information than a categorization would.

5. Implications and Future Work

This work has implications for both computer science educators and researchers studying design. It also raises questions that merit further study. We address these in the following sections.

5.1. Implications for Educators

What significance do these results have for us as educators? The most important lesson learned is that our students might not understand design as well as we would like to think. While there are a number of possible explanations for the observed results, two possibilities are that students have insufficient experience with open-ended problems and with communicating designs. These suggest two practices that educators might do differently.

- Give more open-ended or underspecified assignments. Part of the task would be for the student to deal with ambiguities and determine what (and at what level of detail) is appropriate to deliver.
- Give students the experience of producing designs, and then implementing other student-developed designs. The feedback between designers and implementers would give students practical experience on how to communicate design information.

Both of these would teach students what is important, and what is not, in developing and communicating designs.

Additionally, the academic factors that were most highly correlated with the design categories were the number of CS courses that a student had taken and the time spent on the design task. Grades in Computing Science (CS) courses, on the other hand, were not significantly correlated with the design categories. Given the importance of software design in the computing field, it suggests that software design receives insufficient emphasis in computer science programs.

5.2. Implications for Researchers

These results also have implications for researchers studying design. Primarily, they suggest that a semantic categorization – one that is meaningful in the context of interest—can be used effectively. This work also suggests that much can be learned about design from the produced artifacts, independent of the process. They also suggest that some information can be easily extracted—as with the feature counts and length here—but that getting other useful information requires fairly close analysis of what has been produced.

5.3. Future Study

This work suggests a number of questions to examine further.

Are these results due to inadequate design skills, or a mismatch between the subject and researcher expectations? One author teaching a senior-level Object-Oriented Programming and Design course gave this design brief on an exam without further instructions. All 16 students produced designs in the top three categories, perhaps because they knew what was expected in the context of the course. This suggests a follow-up study could involve a similar design task, where students are given information about how the designs will be evaluated, or other information about what is expected in a ''good'' design.

Are these perceived shortcomings (quickly) cured by experience? Another possible study would involve recent graduates who are working as software developers. Anecdotal evidence suggests that it takes some real project experience for developers to understand the value of proper documentation and formal techniques.

Is it possible to effectively work from particular artifacts and avoid the difficulty of extracting information from free-form text and diagrams? One could give a design task, asking that the produced design include certain thing (such as UML class diagrams)—then base the analysis on the formal artifacts alone.

Are there institutional differences in how students design? Informal examination suggests there are differences here (as were seen in Lister et al., 2004), but making formal comparisons would require more observations per institution (in this study, the average was around 7.5 observations per institution), so that other factors (such as the number of courses taken) might be isolated.

6. Conclusions

The results of this study show that a semantic categorization is both possible and practical in studying designs. Such an approach has one fundamental strength: the categories developed are consistent with the information desired from the data. These categories can be relatively insensitive to stylistic differences, which makes them practical in studying designs from different institutions with different cultural and linguistic traditions. Additionally, these results suggest that useful design data might be extracted from the design artifacts alone. Although categorizing these artifacts can

be somewhat labor-intensive, it is far less so than the alternative of extracting process information, which larger datasets might practically be examined.

In terms of software design, the results of this study show that the majority of graduating students cannot effectively design a software system. Their level of performance is significantly correlated with the number of computer science courses taken, but not with overall performance (as measured by grades) in computer science courses. There seemed to be a lack of understanding about what sort of information a software system design should include, and how to effectively communicate that information. These results suggest possible interventions in practice, as well the need for closer examination of how software design is learned and practiced by students.

Acknowledgments

The authors would like to thank Sally Fincher, Marian Petre, Josh Tenenberg, the Scaffolding workshop participants, and the National Science Foundation (through grants DUE-0243242 and DUE-0122560) for their support and encouragement. Additionally, thanks to the reviewers and participants of Koli Calling, for their questions and suggestions before, during, and after the conference.

Notes

1. These prototypes are actual designs from the dataset, not general descriptions.
2. The Swedish word *Skumtomte* refers to a pink-and-white marshmallow Santa Claus, a traditional Christmas confection. It looks like there is something there, but it is only shaped and colored marshmallow fluff.

References

Atman, C.J., Chimka, J.R., Bursic, K.M., & Nachtmann, H.L. (1999). A comparison of freshman and senior engineering design processes. *Design Studies*, *20*, 131 – 152.

Blaha, K., Monge, A.E., Sanders, D., Simon, B., & VanDeGrift, T. (2005). Do students recognize ambiguity in software design? A multi-national, multi-institutional report. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)* (pp. 615 – 616).

Chen, T., Cooper, S., McCartney, R., & Schwartzman, L. (2005). The (relative) importance of software design criteria. *Proceedings of the 10th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2005)*, June, Monte da Caparica, Portugal (pp. 34 – 38).

DuBoulay, D. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*, 57 – 73.

Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., & Zander, C. (2006a). Can graduating students design software systems? *Proceedings of 37th ACM Technical Symposium on Computer Science Education (SIGCSE 2006)*, March, Houston, TX (pp. 211 – 215).

Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe M., & Zander, C. (2006b). Comparing student software designs using semantic categorization. *Proceedings of the 5th Koli Calling Conference on Computer Science Education (Koli Calling 2005)*, TUCS General Publication No. 41, Turku, Finland (pp. 57 – 64).

Jacob, E.K. (2004). Classification and categorization: a difference that makes a difference. *Library Trends*, *52*, 515–540.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, *36*, 119–150.

McCracken, W.M. (2004). Research on learning to design software. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research.* London: Taylor and Francis group.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education*, *13*, 137–172.

Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T., Chinn, D., Cooper, S., Eckerdal, A., Johnson, J., McCartney, R., Monge, A., Moström, J., Petre, M., Powers, K., Ratcliffe, M., Robins, A., Sanders, D., Shwartzman, L., Simon, B., Stoker, C., Tew, A., & VanDeGrift, T. (2005). Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, *4*, 143–162.

Paper VIII

From *Limen* to *Lumen*: Computing students in liminal spaces

Anna Eckerdal
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Robert McCartney
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT USA
robert@cse.uconn.edu

Jan Erik Moström
Department of Computing Science
Umeå University
901 87 Umeå, Sweden
jem@cs.umu.se

Kate Sanders
Mathematics and Computer Science Department
Rhode Island College
Providence, RI USA
ksanders@ric.edu

Lynda Thomas
Department of Computer Science
University of Wales
Aberystwyth, Wales
ltt@aber.ac.uk

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA USA
zander@u.washington.edu

ABSTRACT

This paper is part of an ongoing series of projects in which we are investigating "threshold concepts": concepts that, among other things, transforms the way a student looks as the discipline and are often troublesome to learn. The word "threshold" might imply that students cross the threshold in a single "aha" moment, but often they seem to take longer. Meyer and Land introduce the term "liminal space" for the transitional period between beginning to learn a concept and fully mastering it.

Based on in-depth interviews with graduating seniors, we found that the liminal space can provide a useful metaphor for the concept learning process. In addition to observing the standard features of liminal spaces, we have identified some that may be specific to computing, specifically those relating to levels of abstraction.

Categories and Subject Descriptors

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

General Terms

Human Factors

Keywords

threshold concepts, liminal space, learning theory

ICER'07, September 15–16, 2007, Atlanta, Georgia, USA.

1. INTRODUCTION

This paper is part of an ongoing series of projects in which we are investigating "threshold concepts": concepts that, among other things, transform the way a student looks at the discipline and are often troublesome to learn. [13] We are interested in identifying these concepts in computer science, understanding how students experience the process of learning those concepts, and designing better ways to help students with this process.

In previous work we describe the theory behind threshold concepts and research related to the theory [6], comparing them to constructivism, mental models, misconceptions, breadth-first approach, and fundamental ideas. Two particular threshold concepts candidates, abstraction and object-orientation, are discussed. With evidence from the literature we argue that these concepts are likely to fullfil the criteria for threshold concepts.

We describe the studies which provide data for the empirical investigation of threshold concepts in computer science in [2]. We used data collected from in-depth interviews of computer-science majors nearing graduation. A preliminary analysis of data from educators provided a number of threshold concept candidates which served as a guideline for the study with the graduating students. This study, described in detail in [2], aimed at identifying possible threshold concepts and we provide evidence of two threshold concepts, object-orientation and pointers.

During the analysis we also found that students report using a wide variety of strategies to make progress in learning these difficult concepts. The re-analysis of the interviews to identify and categorise students' strategies is presented in [11].

The word "threshold" might imply that students cross the threshold in a single "aha" moment, but often they seem to take longer. Meyer and Land [14] introduce the term "liminal space," borrowed from anthropology. *Limen* is the Latin word for threshold, so this literally means "threshold space." Roughly speaking, in our context, the term refers to

the transitional period between beginning to learn a concept and fully mastering it.[1] Meyer and Land's formal definition, discussed below, helped us to formulate the questions that guided our analysis of the data. Our analysis revealed several interesting aspects of how students experience this liminal space.

In Section 2, we give the theoretical background for this work. We present our research questions in Section 4, review some additional related work in Section 3, and describe our methodology in Section 5. In Section 6, we present our results: the different ways in which students experience the liminal space, as shown in our data. We discuss these results in Section 7, and close with our conclusions and future work.

2. THEORETICAL BACKGROUND

Meyer and Land [14] have proposed using threshold concepts as a way of characterizing particular concepts that might be used to organize the learning process. They further develop a theoretical framework, the liminal space, which specifically focuses on the process of learning such concepts.

2.1 Threshold Concepts

Threshold concepts are a subset of the core concepts within a discipline, and are characterized as being [14]:

- *transformative*: they change the way a student looks at things in the discipline.
- *integrative*: they tie together concepts in ways that were previously unknown to the student.
- *irreversible*: they are difficult for the student to unlearn.
- potentially *troublesome* (as in [17]) for students: they are conceptually difficult, alien, and/or counterintuitive.
- often *boundary markers*: they indicate the limits of a conceptual area or the discipline itself.

The idea has the potential to help us focus on those concepts that are most likely to block students' learning. [4]

In our interviews, we have so far found evidence that *pointers* and *object-oriented programming* fulfill the criteria for threshold concepts. [2]

2.2 Liminal space

The term "liminal space" was originally used in anthropology to describe the time during which someone is passing through a rite of passage. [21] When borrowing the term, Meyer and Land list the following defining characteristics. [14] The liminal space is a space in which someone

- is being transformed
- acquires new knowledge
- acquires a new status and identity within the community

The process of being in this liminal space and crossing the threshold may

- take time, and may involve oscillation between old and new states
- involve emotions, of anticipation, but also of difficulty and anxiety
- involve mimicry of the new state

The period of adolescence, for example, has all of the characteristics of a liminal space. In adolescence, an individual is being transformed and acquiring the identity of an adult. This process takes time, involves acquiring new knowledge – how to earn a living, for example – and is often a difficult time emotionally. Adolescents, especially in the early stages, can behave like children one moment and adults the next, oscillating back and forth between the two states. And they learn to be adults, in part, by mimicking the adults around them.

In educational settings Meyer and Land emphasize the transformative character of threshold concepts as the main reason for the liminal space: "owing to their powerful transformative effects" [13, p. 10], and in [14, p. 380] the authors explain: "we see the threshold as the entrance into the transformational state of liminality."

In the learning of threshold concepts mimicry can "involve both attempts at understanding *and* troubled misunderstanding, or limited understanding, and is not merely intention to reproduce information in a given form." [14, p. 377] The authors further relate both mimicry and emotions of difficulties and anxiety to "stuck places." Identifying such stuck places in the learning process can lead to a fuller understanding of the transformation student undergo.

The characteristics of the liminal space given by Meyer and Land, applied to our empirical data, served as starting points for an analysis on conceptual learning in computer science. This research has the potential to shed light on why some students get stuck at the threshold in the process of becoming computer scientists. Meyer and Land write: "liminality, we argue, can provide a useful metaphor in aiding our understanding of the conceptual transformations students undergo, or find difficulty and anxiety in undergoing, particularly in relation to notions of being 'stuck'." [14, p. 377]

3. RELATED WORK

This work fits squarely within the constructivist tradition. Constructivist theory holds that the learner actively builds knowledge. Different theories propose different models for the learner's knowledge: a hierarchy of anchoring ideas [12], schemas [12], and mental models. [8] In each case, however, learning involves adding to or modifying some cognitive structure. To continue the construction metaphor, threshold concepts are keystones, critical parts of the structure that hold the rest together, and the liminal space is the construction site.

No work has specifically addressed the liminal space in computer science. There is a substantial literature on concept learning in general, however. We will restrict ourselves to the work that is most closely related to the defining features of a liminal space.

Perkins and Martin found that students were hindered by what they called "fragile knowledge," that is, when the student "sort of knows, has some fragments, can make some moves, has a notion, without being able to marshal

[1] *Lumen* is the Latin word for the light that we hope students find when they have fully crossed the threshold.

enough knowledge with sufficient precision to carry a problem through to a clean solution." [18, p. 214] Shymansky et al. found support for *oscillation* – "a punctuated, sawtoothed, conceptual growth process" – in a study of a group of middle-school teachers. [19, cited in [20]] In a later study of students, they found that while oscillations were not reflected in the mean ratings, 10 of the 22 individual students did show patterns of progress and regression. [20]

Mimicry is generally considered to be negative – the students are said to be "just mimicking" or "only mimicking" what they have seen. And it can be negative if the student does not progress beyond this point. Hughes and Peiris [7], for example, found a strong negative correlation between course performance and a "surface apathetic approach" to learning to program, in which the students memorize and reproduce what they have seen without any deeper understanding.

On the other hand, if students persist in seeking a deeper understanding while they mimic what they have seen, the practice may be helpful. Muller's work with pattern-oriented instruction suggests this may be the case. [15] In some non-Western educational traditions mimicry is considered to be an important step in learning. [9, 10] In a comparative study of Chinese and Australian students of accounting, Cooper found that "While surface approaches to learning can be associated with mechanical rote learning, memorization through repetition can be used to deepen and develop understanding and help achieve good academic performance." [3, p. 306]

Murphy and Tenenberg address the general question of *whether computer-science students know what they know.* [16] They asked students to predict how they would do on a data structures quiz taken in courses that required data structures as a prerequisite. They found that the students' estimates correlated moderately with their performance, and (interestingly) the accuracy of their estimates improved after the quiz. This was consistent with, or slightly better than, the estimating ability of students in other fields.

Mead et al. propose a method of *organizing a curriculum around what are essentially threshold concepts*, plus some additional "foundational concepts." [12, p. 187] They modify the definition of threshold concepts, requiring only that a concept be integrative and transformative. It seems likely, however, that the other defining features of threshold concepts follow from these two. If a concept integrates other ideas or causes you to see the field in a new way, it may well be troublesome to learn, and you are not likely to forget it. Thus, the set of concepts they focus on likely include all the threshold concepts.

They suggest creating a directed graph with the threshold and foundational concepts as nodes, showing the order in which concepts should be presented in a curriculum. Concept A should be taught before another concept B if it "carries part of the cognitive load in learning it." [12, p. 187] By presenting the concepts in the right order, we may be able to make it easier for our students to learn each of them.

Meyer and Land note that there are inherent conflicts between the use of threshold concepts, particularly as articulated above by Mead et al. as steps in a logical passage through a curriculum, and the more fluid and unordered aspects of liminality on which we will focus here. [14, p. 379-380]

Vygotsky's discusses *the zone of proximal development*, defined as "the distance between the actual development level as determined by independent problem-solving and the level of potential development as determined through problem-solving under adult guidance or in collaboration with more capable peers". [22, p. 86] In other words it is the space from where a student is to where he/she can go next. The student may or may not be in the middle of a transformative learning experience. The zone of proximal development focuses on what is attainable for a student, whereas the liminal space focuses on the transformative aspects of the learning experience.

4. RESEARCH QUESTIONS

This paper addresses two research questions.

1. Can the liminal space as discussed by Meyer and Land serve as a "useful metaphor in aiding our understanding of the conceptual transformations students undergo" [14, p. 377] in computer science?

2. What specific characteristics do we observe in computer science students when they are in the midst of learning a threshold concept, and do these satisfy the requirements of a liminal space?

By addressing these questions, we hope to gain better insights in the complicated process of conceptual learning in computer science. Furthermore we hope to shape the framework for our specific discipline.

We looked for evidence of the different features of a liminal space given in Meyer and Land's definition (above). It follows from the definition of a threshold concept that by learning it, the student is being transformed and acquiring new knowledge. Because threshold concepts are core concepts within a discipline, students learning them can also be said to be acquiring a new identity, that of an insider, someone who understands the central ideas of a field.

We focused, therefore, on the remaining aspects of the definition of a liminal space, looking for answers to the following questions in our interviews:

- Does the process of learning threshold concepts take time? Do the students appear to oscillate between the old and new states (i.e., not understanding and understanding)?

- What emotional reactions do students express?

- Does the process of learning threshold concepts involve mimicry?

We also formulated some questions that are not explicitly addressed by Meyer and Land, but notwithstanding seem to be important for a rich description of the learning of the threshold concepts studied.

- What kinds of partial understandings do students possess within the liminal space?

In the interviews we found that students made a clear distinction between different aspects of the concept they were learning. Most students discussed one or several aspects as troublesome to learn, but different students struggled with different aspects. Each of these aspects is a place where students might become stuck.

The second question we added was:

- Do students know that they have crossed a threshold, and if so, how?

Whether students can tell when they have crossed a threshold is relevant, since the liminal space seems to be accompanied by emotions of frustration or desire to pass through it. If a student thinks he or she has crossed a threshold in learning, even though he or she hasn't, what are the consequences for the motivation to learn?

5. METHODOLOGY

The data used were gathered during a previous study of threshold concepts [2, 6, 11],using semi-structured interviews with 14 students at six institutions in Sweden, the United Kingdom, and the United States. For analysis, the student interviews were transcribed verbatim; where necessary, they were translated into English by the interviewer.

During the analysis of the data we identified two threshold concepts: object-orientation and pointers. [2] This paper continues the analysis by looking into how the idea of a liminal space relates to these threshold concepts. The authors once again read through all the interviews looking for quotes related to liminal space, the resulting selections were then discussed among the authors and related to the discussion of liminal space as described in Section 2.2. The result of this analysis is reported below.

In our interviews we specifically asked the students about concepts they found troublesome to learn. In the present study we have re-analyzed those interviews where students entered deeply into discussions on pointers or object-oriented programming.

6. RESULTS

The analysis was inspired by the goal to investigate the usefulness of the liminal space metaphor in computer science. We call our analysis a triangular conversation, that is an ongoing conversation and negotiation between the researchers, the data, and the liminal space as it is described by Meyer and Land. The questions we asked are inspired by the characteristics of the liminal space, but also by the data, the observed characteristics from the quotes. The answers we found are shaped by the research questions, the data, and our lengthy experiences as teachers in the subject domain.

6.1 Partial understanding

Since the liminal space for a concept is the time when the student is trying to attain a concept but has not yet succeeded, it should be characterized by partial attainment of a concept. When looking at the quotes relating to partial understanding, theoretical as well as practical, we see a number of common themes emerge. Students identify a number of different sorts of understanding including abstraction of the concept, concrete implementation of the concept, and the ability to go back and forth between the two. The observed understandings could be placed into these general categories:

- An abstract (or theoretical) understanding of a concept;
- A concrete understanding—the ability to write a computer program illustrating the concept—without having the abstract understanding;
- The ability to go from an understanding of the abstract concept to software design or concrete implementation;
- An understanding of the rationale for learning and using the concept; and
- An understanding of how to apply the concept to new problems—problems beyond those given as homework or lab exercises.

Abstract understanding

Some quotes showed that the abstract concept was not yet attained. This quote showed a confusion between the notions of *class* and *object*:

> *Subject 9:* I can still remember that I tried to do operations on the classes that I think I can really remember, but I think I was trying to let the lamp shine or don't shine by doing something with the lamp class instead of with the lamp object.

Another showed the difficulty of learning pointers was tied in with other unlearned concepts:

> *Subject 4:* You know, and I'm not sure what I didn't understand, because there was plenty of other things that we were doing at the same time, like recursion and inheritance, that also used pointers. Recursion was another huge stumbling block for me. And so taking a pointer and throwing it in with a recursive function - [laugh] - I felt like I was, you know when you stand in front of those mirrors in a dressing room, the ones that are in front of you and on the sides, and you see reflections and reflections and it never ends? That's what I felt like with pointers and recursion.

Some showed that some understanding had been successfully attained:

> *Subject 6:* But now I can ... I'm able to see how the classes are related, I guess. How they're related and which classes share information.

Some students were knowingly striving for a deep understanding:

> *Subject 5:* Why and how it [OOP] should be used. I have a background that is very much procedural imperative ... programmed Basic, Assembler and Pascal since the middle of the -80s ... so it was a rather high threshold to, not to learn how to use it, to get it to work, but to use it the right way ... I thought it was very elegant but it took probably several years before I saw the really elegant solutions ...

Concrete understanding without abstract

Some students were able to work with object-oriented concepts at a concrete level without a theoretical understanding:

> *Subject 9:* ... I'm pretty good at Java, but the interface concept is little strange. Abstract class

and interface and stuff like that, ehh, is rather complicated. Ahh, specially interface [giggle]. And to explain that to someone, I don't think I can do it, but I can use the term and I can use interfaces.

Relating the abstract concept to implementation or design

Students commonly mentioned that they had a theoretical understanding, but were unable to translate that understanding to something less abstract. Specifically, many students discussed their inability to use their abstract understanding to produce a concrete implementation:

> *Subject 7:* There's just some aspects to it that just seem to remain kind of mysterious to me at the programming level. Not the concept level, not the theory level, not the technology level but at the kind of code nuts and bolts level ... It's not that I don't understand what I'm trying to accomplish it's just getting the syntax of the details right ... I'm a lot better in Java because I don't have to deal with the syntax of the details. I can only deal with the concepts.

> *Subject 8:* the abstract understanding is something you learn by education, by reading, you can learn that in class, but the understanding of actually applying it to programs you can't, you must, you must learn it by, by, by using it ...

Other mappings proved difficult, such as the application of the abstract understanding to design, which is less concrete than implementation:

> *Subject 8:* ...but the harder thing is actually to create what should be an object and what should not be an object and, what classes should I have for these things and, and that is the most ...

A similar observation is the following:

> *Subject 9:* ... So it wasn't like a big struggle to understand the difference between class and object for me actually. But it can be when you're designing a program ... To know where to stop doing the classes and start doing the objects. That ... that's actually something you can think about today, as well.

Rationale

Another aspect of understanding a concept is understanding its rationale–why you would want to know and use this concept. Students reported feeling the lack of this understanding as they were learning:

> *Subject 3:* My thoughts were that I didn't understand why we needed pointers when references worked perfectly well beforehand. I didn't understand the power of pointers and I guess I just didn't see the purpose of declaring variable `int*`[2].

> *Subject 5:* I found it difficult during the first ... the first course when I encountered it, I couldn't see the use of it, except that you could get some kind of encapsulation.

Application

Another way of understanding a concept is to understand how to apply it to new problems, not just in the related assignments.

> *Subject 2:* You understand how a theory works but how do you take that theory and how it works and apply it to a practical sense? I think that is one of the hardest leaps to make.

> *Subject 8:* ... it took a long time to understand how object oriented programming works, but then once I understood it more or less, the basic concept, I still couldn't use it, it wasn't usable because I didn't know what to apply to my problems.

6.2 Temporal Aspects and Oscillation

We examined how long students spend in the liminal space. Since we started our discussions by asking students about places where they were stuck, it is not surprising that all of our students emphasized the prolonged process required to learn threshold concepts.

Students and faculty alike will often talk about having an "aha" moment. While this might imply a sudden insight, this moment frequently is preceded by either a long time in the liminal space or a depth of understanding in a related area. Subject 7 implies a lengthy journey through the liminal space:

> *Subject 7:* It unwound or wound it printed out statements but I still didn't understand it very well. It really honestly wasn't until I got to your class that the light kind of came on and the idea of doing the checks up front and sort of assuming it's going to do what you tell it to do.

Subject 5 describes a deep understanding before the "aha" moment:

> *Subject 5:* A friend that showed me some kind of interpreter for some small little ... well, a model of a computer, where he [...] took the instruction object and told it 'run', [...] then I got some kind of small aha-experience, 'perhaps you can do it that way instead of doing it in some more tiresome way', the way I should have done it myself.

Many students mentioned a prolonged time in the liminal space. Across the board, the time to gain understanding was lengthy:

> *Subject 13:* I think there was definitely a point where I definitely got the understanding, whether I was still confident in doing it, that probably took a lot of time.

> *Subject 2:* ... when I finally did make the understanding, which actually took about two to three years.

[2] `int*` is a reference to how a pointer to an integer is declared in C-like languages

Subject 6: I think it is something that takes at least a couple of semesters. I mean, unless you've had prior experience, I just don't see.

Subject 5: Object oriented programming was one thing for example that took a long time before ... it clicked. [...] It took ... perhaps two years before it was completely in-place ...

Students seemed to not only spend time learning the concepts, but they also demonstrated understanding that the prolonged process was necessary for learning. Subject 4 knowingly gives a lot of time to the learning process, while Subject 7 implies it was natural to not understand *yet*:

Subject 4: So, I had a lot of time to spend, you know, brain resources to spend understanding, you know, stuff that's pointing and how you dereference it.

Subject 7: So everybody - it was just sort of like they were talking a language I wasn't fluent in yet.

Students also implied oscillation in their learning. They describe the nature of going back and forth between knowing and not knowing, thinking that they know it, but realizing they're not there yet:

Subject 4: It was clear to me, it just seemed like while I was in the thick of it I would forget. I spent a lot of time lost in the - it was that forest for the trees. I don't know. Lost in the jungle.

Subject 6: ... with object-oriented [...] I think you understand the basic - you have the concept of it. But I ran into certain things with classes where I didn't have access to that particular class and I'm thinking, What's the problem here? [...] So I did understand but I have run into problems and it did kind of go back to objects and how they're relating.

Subject 9 nicely summarizes the oscillation between the knowing and the not knowing:

Subject 9: ...most of the time it's just iterates on the outside of the knowledge spiral. [...] I have to refresh the knowledge I learned recently more often, but some things I have to go back and refresh maybe the real basics of what it's all about. Not that I have forgotten it but to get a deeper understanding.

6.3 Emotional reactions

Meyer and Land refer to the liminal space as "problematic, troubling, and frequently involv[ing] the humbling of the participant." [14] They also warn that students may experience "difficulty and anxiety" in relation to learning threshold concepts. We examined our student quotes from this perspective, to see if we could find evidence of emotionally laden terms.

Students frequently mentioned that they found that learning threshold concepts was frustrating:

Subject 3: Felt? I think I felt frustrated. My thoughts were that I didn't understand why we needed pointers when references worked perfectly well beforehand.

Others referred to feelings of depression:

Subject 13: During ... well if I found it difficult then I would probably mope slightly for a while and then got down to it.

There was evidence of students feeling humbled:

Subject 2: Another thing that was very frustrating. I'm usually quick to understand things.

Subject 7: It just seems like it's been such a long and horrible road over pointers and that object oriented thing. That's just been my nemesis the whole way through and I don't remember anything else being that difficult.

Students themselves note that there was a certain mystique around becoming a programmer or understanding a concept:

Subject 7: The class idea was just really mysterious.

Subject 4: ... it seemed like something really hard. Like if you're extremely smart then you can program, you know. All computer geeks are really smart and they can program. That's my sort of opinion of it before I started. Something that was magical and hard.

When students eventually grasped a concept, they were transformed and excited:

Subject 4: While I was stuck they [pointers] were a nightmare and I hated them. After I figured them out, they were very cool and useful. And I could see why you would want to have them.

Subject 6: And then when I do get it to work, it's almost like these people that run a 25-mile marathon just for, like, that high or whatever. I get that when I solve the problem. I get real souped, screaming in my room.

Confidence can be seen as an emotion with a fairly complex relationship with liminal spaces–being stuck can lower it, but having it can make it easier to get unstuck. Student 9 describes his or her feelings about the importance of having some prior knowledge in programming:

Subject 9: I got the aha experience again and that just was like if I know a little then I can, eh, jump in everywhere and catch up from there. [...] And that's really important to know, or to feel, that you can catch up. [...] it's not impossible.

The same student also emphasizes the importance of knowing "how to study" in terms of be able to use different online resources and IDEs:

Subject 9: That helped in Java doc and API on the Internet and the, aha experience again, that how to use it [...] that you could actually go there and see how you should do with any question you have and also seek information on other eh on Google for example on the Internet how to solve a specific problem, problem in programming. Eh, that have really helped me a lot, Ahh, the confidence that I can do it with, eh, help of Internet. [...] when I hear of a new concept it's just to see on the website to see what they mean and how they, how you should use it and you use it. Ah, and that's a really big threshold to come past.

6.4 Mimicry

During the interviews some of the students mentioned that in beginning to learn a subject they "imitated" someone or some existing code. Subject 9 discusses starting object-oriented programming:

Subject 9: And then ... when ... to learn something from an example, for example, it had to be exactly almost the same example as the thing you are trying to solve. You are trying to find exactly the information how to solve this problem in the textbook always search in the textbook.

and:

Subject 3: I think if a person can see the pattern, I think I'm no different from anyone else. If I can see the pattern, I can generally, I can take a technique and I can go home and figure it out if there's a pattern to it.

Others indicated that it was a big help in the beginning to have step-by-step instructions to follow:

Subject 7: I think the idea - and one of the things in your teaching - one of the things about your teaching is that you tend to give a procedure and I think I don't believe in - you actually in some cases give a list of things. Step one, step two like on recursion. I don't think in other classes that we've been that procedure oriented. Maybe we've talked more about the idea and the concept and the whatever, but it really helped.

Even if a behavior like this might seem counter-productive, "the students are here to learn how to do things themselves, right", it is important to realize that for the interviewed students the "mimicry" seemed to be just a stepping stone in their further learning. Here is another example of a student who started by mimicking and then progressed:

Subject 9: In the beginning I tried to look it up in the textbook and find the exact example how to solve this instead of, eh, while during the process I found that an IDE can help me when I, when I press the point it gets a list of everything that's possible to do with that object, and if you write the class name, you, you get some sort of error message, it probably meant that instantiation of this object. Ah, so it helps ...

Interviewer: So, in this way ... are you using this IDE. Your understanding of these concepts changed?

Subject 9: They improved, yes.

In some cases, however, students did not progress beyond mimicking:

Subject 7: I have so much trouble with that overload asterisk and there's that – is it asterisk ampersand symbol or whatever. Never got that. Never had a clue. I just copied it. Yeah, it really gave me trouble. Just looking at would just sort of freeze me.

6.5 Crossing the threshold

Students in the interviews discuss object-oriented programming and pointers from an illuminated perspective of having passed the liminal space. This is expressed in different ways. Sometimes the descriptions of the experience of passing through the liminal space is emotional:

Subject 2: It took a lot of just practicing and just repeating. It's to the point where when you see it you wouldn't be kind of intimidated. You would already say okay I know what I can do with this.

One student discusses the emotions that characterizes his or her conviction of having passed the liminal space, and the previous emotional conviction of not having passed:

Subject 6: But I just remember at that moment like it just kind of made, I don't know, made sense, I guess. I don't know what about it made sense. [...] I mean, I did get it before. I saw what was going on. But I just didn't feel like I had the control, I guess, till I saw it.

Some students describe their conviction of having passed the liminal space as being able to visualize their understanding:

Subject 2: But the basic idea of passing by reference or value; no, once I understood that I – every time it's mentioned I immediately know and understand – I can see a picture – a diagram in my head of what I'm supposed to do.

Subject 7: I remember in the final I looked at a problem that you wrote and I saw recursion ... I remember it was a tree and I remember looking at it and as I said some people see black and white, some people see color. It was like I saw color. Oh, you can solve this with recursion. It's a tree. I can solve this recursively and here's this relationship. [...] That was kind of like, "Whoa." I actually saw it and that was pretty exciting.

Other students describe their conviction of knowing the concepts on the foundation of mastering the handicraft of programming:

Subject 13: And after ... its like you said before it was one of those things like riding a bike, isn't it.

And another student says:

> *Subject 2:* And then after it's almost like it's a tool and you don't even think about using it. You say I need to do this. Okay, done. [...] And it's a seamless integration. It's just there it is. And you don't - it's almost like you don't even think. Like when you - right now I have to declare an integer. I don't think about how I do it or how to syntax. I just type it away. It's almost like a memory response.

The same student contrasts his or her experience to how it was before the passage of the liminal space:

> *Subject 2:* So I think it comes from a point of being completely lost and just randomly guessing and hoping your guessing is good. To a point where you're confident with using that and you may not want to use it as much as you would something else you're more confident with ...

Having passed the liminal space does not always mean that there is never a need for going back. The students distinguish however between not understanding and practicing for a better understanding, and between understanding but still needing to practice syntactical details of the programming language:

> *Subject 3:* After a certain length of time, yeah, sure, I have to review stuff.
>
> *Interviewer:* Well, you review it to program, but conceptually?
>
> *Subject 3:* No, I understand it. It's something I do get.

Similarly:

> *Subject 2:* I would have to look up the syntax and possibly get a very brief example just to remind myself that's how the pointer works. Okay, done. Then the memory jog hits me and I'm good.

An interesting question arises when studying students who claim they have passed through the liminal space. Are the students' views are in line with the educators' view of what is required for a "good" understanding of the concept? Are there students who believe they have passed the liminal space, when they, according to the course requirements, have not? And, on the other hand, are there students who believe they have not passed the liminal space, while educators would say they have? Students from our study illustrate this:

> *Subject 9:* So then, and still, I, I mean that I'm pretty good at Java, but the interface concept is little strange. Abstract class and interface and stuff like that, is rather complicated. 'specially interface. And to explain that to someone, I don't think I can do it, but I can use the term and I can use interfaces.

And later in the interview:

> *Subject 9:* I think I should know why information hiding is important but I can't think of it now ...

It can be questioned whether the student has passed the liminal space or not since the concepts the student fails to understand are central to the object-oriented paradigm.

Another student demonstrates his or her understanding of object-oriented programming, and yet says

> *Subject 5:* object oriented programming was one thing for example that took a long time before...it clicked. [...] It took...perhaps two years before it was completely in-place...and it's really nothing that I've really understood even today.

Reading the transcript as educators, we believe the student has a good understanding, and still he or she is not convinced of having passed the threshold.

7. DISCUSSION

Students certainly describe the features that define liminal space according to Meyer and Land. Our analysis has raised a number of interesting observations and questions.

First, we saw different partial understandings of students during the liminal space. Students, at least in retrospect, show an appreciation that full understanding includes the theoretical and the practical: an abstract understanding, the concrete ability to implement, being able to go from the abstract to the concrete, the underlying rationale, and how to apply the concept. The need to attain all of these somewhat independent understandings explains why students get stuck at different places, and why the path through this space is not a simple linear progression. That we commonly observed the particular partial understanding of not being able to translate from an abstract understanding to concrete implementation or design may be specific to computing as a discipline, a question worth deeper investigation.

This observation in the data seems to be in contradiction to some well-established taxonomies, e.g., Bloom's taxonomy. [1] Bloom's taxonomy defines goals for learning in levels. We, on the other hand, have observed that the different aspects of understanding occur in parallel. In fact, the need to go back and forth between the theoretical and the practical seems as important as the parts themselves.

We want to emphasize that the identified partial understanding, the parts or aspects, refers to the whole learning process including both the theoretical and practical understanding. These parts are often developed in parallel. Mastering a concept requires an abstract understanding, a concrete ability to implement it, the rationale behind using it, and an understanding of how to apply it in practice. The different "routes" students take correspond to different patterns of development: some students claimed that they had no problems with the theory, but struggled to see the rationale behind the concept, while others had difficulty applying the concept to new problems.

Second, when considering the question – "Does the process of learning threshold concepts take time?" – the answer seems to be a clear *Yes*. All of our subjects at some point discuss the lengthy process of learning. What we found interesting here was that whether acknowledging that learning occurs as a spiral action or as feeling lost in a jungle, all of our graduating students admit and accept that learning computing concepts takes time. Some students take time with the theoretical aspects, while others spend more time learning how to implement the concept. This may be a major roadblock to first-year students who typically have not

yet learned about the time-consuming nature of learning, particularly in a technically demanding field such as computing.

If so, then one thing educators can do is to support students during the experience that learning takes time. While many of us attempt to do this indirectly with our assignments and labs, is there something we can directly do about this? How can we instill the notion that the time-consuming nature of learning is normal?

This should also be taken into consideration by educators when they meet novice students. The insight that these students lack the experience that learning takes time might help educators to better understand and cope with the difficulties novice programmers demonstrate.

Third, we found that there was no lack of emotional reactions while learning threshold concepts. As our interviews show, students exhibit very strong feelings. This presence of strong emotion in students discussing the field of computing is rarely mentioned in the literature, but as CS educators, many of us have had the experience of students telling us that they "hate programming." Despite purists' belief that computing concepts should not be anthropomorphized [5], our students personalize threshold concepts, and say they *hate* or *fear* them. They also exhibit feelings of euphoria when they emerge on the other side of the threshold.

We suggest that instead of dismissing students' emotional reactions, as teachers and professionals we should recognize that they are normal and desirable. We need to consider how we can create a learning environment where the feelings that programming is hard, magical and frightening are handled, and students move through them rather than give up.

Fourth, many students state that at some stage during the learning process they mimic what others have done without exactly understanding what they are doing. For some teachers mimicry is an undesirable action; students are supposed to always understand what they are doing and to "just mimic" someone is a failure. We suggest that teachers look at mimicry in another way. Although some students do not progress past mimicry, it can be a step to gaining a full understanding of the subject. Meyer and Land acknowledge this when they write

> ...students might well adopt what appears to be a form of mimicry as a serious attempt to come to terms with conceptual difficulty, or to try on certain conceptual novelties for size as it were. We would not want to belittle or dismiss such responses as they may well prove to be successful routes through to understanding for certain learners. [14, p. 383]

Our original interviews did not pursue the idea of mimicry in depth, but the mixed results here suggests that further study of the role of mimicry in learning programming (and computing in general) is warranted.

Fifth, the question "Do students know that they have crossed a threshold, and if so, how?" has no clear answer. We have identified different ways students express their belief that they have passed through the liminal space. The descriptions are often vivid and illustrate the students' experiences of a transformation. Yet, while students express that they understand a subject, the evidence suggests that they might be wrong. What are the consequences if students think they have passed through the liminal space, when they have not? And, on the other hand, how does it affect students if they believe themselves not to have reached desired understanding, when the educator judges that they already have passed the liminal space?

8. CONCLUSIONS

In this study, we examined Meyer and Land's notion of liminal spaces in the context of learning concepts in computer science. We applied the theory to our interview data, as a grid to highlight certain aspects of the learning experience. The result of the analysis revealed a broad and rich picture of the students' learning experiences. The picture indicates a transformative experience when learning threshold concepts. The learning process is experienced as a complex whole by the students, and therefore difficult to fully understand. Using the idea of liminal space as an analytic tool provides a way to separate out several important aspects of the learning process. Each aspect is individually interesting as is their complex interaction.

Addressing our research questions, we found

1. Liminal spaces provide a useful metaphor for the concept learning process, at least for transformative concepts. The absence of a single path through, the fact that these changes can take time, the emotional reactions of the students, and the students' use of mimicry as a coping mechanism: these characteristics seem to capture much of the learning experience.

2. In addition to observing the "standard" features of liminal spaces, we have identified some that may be specific to computing. The kinds of partial understandings observed—specifically those relating to levels of abstraction—are closely tied to what computer scientists do.

The particular students' emphasis on the difficulty in going from the abstract to the concrete is quite interesting, and seems counter-intuitive given the way we teach computing. The emphasis in computer science education is on teaching students to abstract away from the details, but the problems observed here are in moving in the other direction.

The most important practical observation from this work may be that different students take different routes through the liminal space, with the possibility of getting stuck at multiple places. This suggests that there is no fixed order of topics that best serves all students, rather instruction should be flexible enough to accommodate individual students. Knowing what aspects of a concept are necessary to gain full understanding, particularly those concerning different abstraction levels and the mappings from more to less abstract, could help here.

This work suggests a number of questions that deserve further investigation. Would we get similar results if we interviewed students while they were still in a liminal space, rather than after they have attained understanding? Would we see differences if we interviewed novices rather than graduating seniors? Would other transformational concepts in computer science—those less tied to programming, for example—show similar partial understandings? How do students use mimicry when trying to learn, and when might it be effective?

ACKNOWLEDGMENTS

The authors would like to thank Mark Ratcliffe and Jonas Boustedt, who participated in the design, data collection, and initial analysis of the threshold concept interviews. Thanks also the Department of Information Technology at Uppsala University for providing us with workspace and facilities in Uppsala, and to Sally Fincher, Josh Tenenberg, and the National Science Foundation (through grant DUE-0243242) who provided workspace at the SIGCSE 2006 conference in Houston. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Uppsala University.

9. REFERENCES

[1] B. Bloom(Ed.). *Taxonomy of Educational Objectives, the classification of educational goals - Handbook I: Cognitive Domain*. McKay, New York, 1956.

[2] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Threshold concepts in computer science: do they exist and are they useful? In *SIGCSE-2007*, pages 504–508, Covington, KY, March 2007.

[3] B. J. Cooper. The enigma of the Chinese learner. *Accounting Education*, 13(3):289–310, 2004.

[4] P. Davies. Threshold concepts: how can we recognise them? 2003. Paper presented at EARLI conference, Padova. http://www.staffs.ac.uk/schools/business/iepr/docs/etcworkingpaper(1).doc (accessed 25 August 2006).

[5] E. Dijkstra. On the cruelty of really teaching computing science. *Commun. ACM*, 32(12):1398–1404, 1989.

[6] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander. Putting threshold concepts into context in computer science education. In *ITiCSE-06*, pages 103–107, Bologna, Italy, June 2006.

[7] J. Hughes and D. R. Peiris. ASSISTing CS1 students to learn: learning approaches and object-oriented programming. In *ITiCSE-06*, pages 275–279, Bologna, Italy, June 2006.

[8] P. N. Johnson-Laird. *Mental models: towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983.

[9] D. Kember. Misconceptions about the learning approaches, motivation and study practices of Asian students. *Higher Education*, 40:99–121, 2000.

[10] F. Marton, D. Watkins, and C. Tang. Discontinuities and continuities in the experience of learning: An interview study of high-school students in Hong Kong. *Learning and Instruction*, 7(1):21–48, 1997.

[11] R. McCartney, A. Eckerdal, J. E. Moström, K. Sanders, and C. Zander. Successful students' strategies for getting unstuck. In *ITiCSE-07*, pages 156–160, Dundee, Scotland, UK, June 2007.

[12] J. Mead, S. Gray, J. Hamer, R. James, J. Sorva, C. St. Clair, and L. Thomas. A cognitive approach to identifying measurable milestones for programming skill acquisition. In *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 182–194, New York, NY, USA, 2006. ACM Press.

[13] J. Meyer and R. Land. Threshold concepts and troublesome knowledge: Linkages to ways of thinking and practising within the disciplines. ETL Project Occasional Report 4, 2003. http://www.ed.ac.uk/etl/docs/ETLreport4.pdf.

[14] J. H. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49:373–388, 2005.

[15] O. Muller. Pattern oriented instruction and the enhancement of analogical reasoning. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 57–67, New York, NY, USA, 2005. ACM Press.

[16] L. Murphy and J. Tenenberg. Do computer science students know what they know?: a calibration study of data structure knowledge. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 148–152, New York, NY, USA, 2005. ACM Press.

[17] D. Perkins. The many faces of constructivism. *Educational Leadership*, 57(3):6–11, 1999.

[18] D. N. Perkins and F. Martin. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[19] J. A. Shymansky, G. Woodworth, O. Norman, J. Dunkhase, C. Matthews, and C.-T. Liu. A study of changes in middle school teachers' understanding of selected ideas in science as a function of an in-service program focusing on student preconceptions. *Journal of Research in Science Teaching*, 30:737–755, 1993.

[20] J. A. Shymansky, G. Woodworth, O. Norman, J. Dunkhase, C. Matthews, and C.-T. Liu. Examining the construction process: a study of changes in level 10 students' understanding of classical mechanics. *Journal of Research in Science Teaching*, 34(6):571–593, 1997.

[21] V. Turner. *From Ritual to Theatre: the human seriousness of play*. Performing Arts Publications, New York, 1982.

[22] L. Vygotsky. *Mind in Society*. Harvard University Press, Cambridge, Mass., 1978.

Paper IX

Ways of Thinking and Practising in Introductory Programming

Anna Eckerdal
Division of Scientific Computing,
Department of Information Technology,
Uppsala University, Uppsala, Sweden
Anna.Eckerdal@it.uu.se

Abstract

In computer programming education it is generally acknowledged that students learn practical skills and concepts largely by practising. In addition it is widely reported that many students face great difficulties in their learning, despite great efforts during many decades to improve programming education.

The paper investigates and discusses the relation between novice computer programming students' conceptual and practical learning. To this end the present research uses Ways of Thinking and Practising, WTP as a conceptual framework. In the present research Thinking is discussed in terms of students' learning of concepts, while Practising is discussed as common novice students' programming activities.

Based on two empirical studies it is argued that there exists a mutual and complex dependency between conceptual learning and practise in students' learning process. It is hard to learn one without the other, and either of them can become an obstacle that hinders further learning. Empirical findings point to the need to research the relationship between conceptual understanding and practise to better understand students' learning process.

The paper demonstrates a way to research how students' learning of practise and concepts are related. Results from a phenomenographic analysis on novice programming students' understanding of some central concepts are combined with an analysis based on elements from variation theory of the students' programming activities. It is shown that different levels of proficiency in programming activities as well as qualitatively different levels of conceptual understandings are related to dimensions of variation. The dimensions of variation serve as interfaces between the activities and conceptual understandings. If a dimension is discerned, this can facilitate coming to richer conceptual understandings *and* learning additional activities.

1 Introduction

As a teacher, both at upper secondary school and at university level, I have noticed how students learn through their thinking and reflection paired with practical work in the lab. In my teaching practise, it occurred to me that neither "learning by doing" nor "learning by thinking" seemed solely to fully cover the complex process of learning to program.

In this paper I investigate how learning of concepts[1] and learning of practise are related in novice programming students' learning. Practise is discussed in this paper with a focus on learning to *do* practise, while learning *through* practise is discussed merely as a background. The investigation is based on data from two empirical studies. Inspired by some salient findings in the data, the research question posed is:

How are conceptual learning and practise related in programming students' learning process?

As a conceptual framework I use Ways of Thinking and Practising, WTP, as introduced in the ETL project[2] (Entwistle, 2003) and further developed by McCune and Hounsell (2005). WTP highlights the fact that competence in a subject area involves the ability to master certain subject-specific ways of Thinking *and* Practising. WTP draws the attention to necessary understanding and skills in the area. These two aspects are very apparent in the subject area of computer programming, where good conceptual understanding and practical skillfulness are interwoven parts of the learning goals.

I have used a phenomenographic approach (Marton and Booth, 1997) to study students' understanding of fundamental concepts in introductory programming (see Section 5.1 for details). The phenomenographic analysis reveals qualitatively different categories of ways in which the students understand those concepts.

In a subsequent analysis of the practise I have used elements from variation theory (Marton and Tsui, 2004) to study students' learning of common lab activities (see Section 5.2 for details).

The two analyses made it possible to relate qualitatively different conceptual understanding to typical practical activities involved in novice computer programming. My analysis shows that activities at different levels of proficiency relate to different categories of understanding of the concepts through *dimensions of variation*, which serve like interfaces between different conceptual understandings and practises in students' learning.

An analytical model is proposed, where the concept dimensions of variation is used not only in relation to qualitatively different conceptual understandings, but also in relation to different levels of practical proficiencies. In this way the

[1] I will refer to learning of concepts as *conceptual learning* and understanding of concepts as *conceptual understanding* in the rest of the paper.

[2] Information about the ETL project, Enhancing Teaching-Learning Environments in Undergraduate Courses, is found at http://www.etl.tla.ed.ac.uk/ (Retrieved 2008-12-27)

traditional use of phenomenography and variation theory is extended to include not only conceptual understandings, but also the practise, and specifically, how these relate in the learning process. The present empirically based research contributes to a better understanding of the complex relation between conceptual understandings and activities in introductory programming students' learning.

The outline of the paper is as follows: the background of the research is presented in Section 2. This includes a discussion on what WTP means in the programming discipline, a description of the empirical studies, and related work. Section 3 presents the research approaches used in the present research, while the empirical data underpinning the research are discussed in Section 4. Section 5 proposes an analytical model that sheds light on the complex relation between learning of practise and learning of concepts. Section 6 presents conclusions and future work.

2 Background

The goal of the present study is to explore the roles of and relation between Thinking and Practising in novice programming students' learning process. To this end some salient findings from two empirical studies are highlighted by means of the conceptual framework WTP. This section first discusses WTP as it may apply to the programming discipline, then the empirical studies that the research builds upon, and finally research related to the present work.

2.1 Ways of Thinking and Practising in the programming discipline

What does WTP mean and involve when applied to the programming discipline? WTP is a very wide framework. It has been described by McCune and Hounsell in the following way:

> the richness, depth and breadth of what students might learn through engagement with a given subject area in a specific context. This might include, for example, coming to terms with particular understandings, forms of discourse, values or ways of acting which are regarded as central to graduate-level mastery of a discipline or subject area. [...] WTP can potentially encompass anything that students learn which helps them to develop a sense of what it might mean to be part of a particular disciplinary community (McCune and Hounsell, 2005, p. 257)

McCune and Hounsell's discussion implies that the WTP framework involves a whole subject-specific culture. The present research does not aim to fully cover WTP in all programming communities. Some aspects will be discussed, and these aspects will later be related to some aspects of WTP found in the empirical data.

The role of Thinking in the programming discipline

McCune and Hounsell (2005) discuss *Thinking* in the WTP framework in terms of students' learning experiences "through engagement with a given subject area in a specific context" which might include for example "coming to terms with particular understandings, forms of discourse, values" (McCune and Hounsell, 2005, p. 257). Conceptual understanding is one aspect of "coming to terms with particular understandings" and will be the focus of the present discussion of Thinking in computer programming.

Some of the central concepts within the programming discipline are commonly introduced early in programming education and many novice students find them difficult to learn (Eckerdal, 2006).

Thinking, as it is discussed above, appears in all phases of software development for example problem analysis, software design, implementation and testing. This presupposes good understanding of underlying concepts. In object-oriented programming, analysis and design involve for example identifying objects in the problem domain, while implementation means coding the design in a programming language to express the underlying concepts, and testing among other things involves checking the software with reference to these concepts. Thinking in this context thus means the way to think of and understand concepts that enables analysis, design, implementation and tests.

The role of Practising in the programming discipline

The practical side of programming becomes apparent when software development is discussed. Software development is a kind of problem solving process, traditionally divided into several phases: problem analysis, software design, implementation and testing. Programmers often work iteratively among these phases, going back and forth as the work advances. All phases have both practical and theoretical aspects, and some of the practical aspects will be discussed here.

The problem analysis and design phases depend on the character of the problem, and the context where the program will be used. The practise involved in analysis and design requires skills in such diverse areas as programming paradigms and languages, systematic approaches to analysis and design, knowledge of design languages like UML[3] with specific software to produce such design, and hardware knowledge and more.

The implementation and testing phases of software development require for example practical experience in use of several programming languages with appropriate IDEs[4]. The ability to read, write and test code, and knowledge about effective problem solving strategies, are other fundamental aspects of practise important for a programmer to master.

[3]UML, Unified Modeling Language "is a standardized specification language for object modeling. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model." Retrieved 2007-10-08 from http://en.wikipedia.org.

[4]IDE, Integrated Development Environment, are software tools used for programming development, often integrating for example editor, compiler, and debugger.

In this paper I will focus on certain aspects of Practising relevant for novice students.

Thinking and Practising as ways to abstract a problem
What is then the core of WTP from a computer science perspective? Or more precisely, how does a computer scientist approach a programming problem? A computer scientist thinks in terms of *abstractions* (Kramer, 2007). These abstractions are often of two kinds. *Data structures* are abstract models of something, maybe in reality, or concrete in the sense that they concern basic, efficient ways to store data in computer memory. Abstraction is also required in *algorithms* expressed for example through control structures. When a computer scientist approaches a problem, he or she must distinguish between on the one hand data and data structures, and on the other hand how these can be used in algorithms with the help of control structures. In this way problem solving is to formulate algorithm-resembling systematic procedures, where a problem is abstracted into a computer-oriented solution. Abstraction is carried out through Thinking, as described above, and the results of the process of abstraction are concretised through Practise. Without Practise, software development will remain castle in the air, and without basic conceptual understanding, efficient programming solutions are hardly possible to accomplish. Practise and Thinking are thus interrelated and inevitably interwoven in the process of abstracting and implementing a solution to a programming problem.

2.2 The studies

In order to study the role of Practise and Thinking in novice computer programming learning, two empirical studies have been carried out. In the following, they will be referred to as Study 1 and Study 2, respectively.

The majority of the students who take an introductory programming course at Uppsala University are not computer science majors. Thus, in Study 1 fourteen first year students from a study program in Aquatic and Environmental Engineering were interviewed. The main focus of Study 1 was on students' understanding of central concepts. The interviews were transcribed verbatim and translated to English where necessary. The interviews were analysed mainly with a phenomenographic research approach (Marton and Booth, 1997). Results of analyses performed on data from this study are reported by Eckerdal and Thuné (2005), Eckerdal and Berglund (2005), Eckerdal (2006), and Thuné and Eckerdal (2009).

Study 2 was multi-national and multi-institutional. It aimed to investigate threshold concepts (Meyer and Land, 2005) in computer science. Seven researchers from universities in Sweden, the UK and the USA performed semi-structured interviews with a total of 16 graduating computer science students. The interviews were transcribed verbatim, and were translated to English where necessary. These interviews have been analysed from three different angles. The first analysis aimed to identify threshold concepts in the discipline. The second looked at the parts of the interviews where the students discussed strategies to

get unstuck. The last analysis took a theoretical standpoint. The investigation aimed to analyse what *liminal space* (Meyer and Land, 2005) means and involves in computer science, and thus searched for evidence of several characteristic aspects of such learning experience. Results from these analyses are reported in Boustedt et al. (2007), McCartney (2007), and Eckerdal (2007).

Although both Study 1 and Study 2 where constructed to focus on students' learning and understanding of concepts, the students in both studies talked much about the role of practise in their learning. This is reflected in the publications where McCartney et al. (2007) discuss practise in terms of students' strategies to get unstuck, and where Eckerdal et al. (2007) discuss different theoretical and practical parts of students' learning process.

2.3 Related work

In the computer science community it is generally acknowledged that learning to program requires learning concepts as well as practise. In order to better understand the complex relationship between the two, related work from several related areas has been investigated.

The relation between and nature of Thinking (concepts) and Practising in learning has long been debated and researched by philosophers and others, especially educationalists, interested in learning and learning goals. See Molander (1996, and references therein) for an overview from the area of philosophy. Since my research interest is in computer science education, this section will look at related work from educational research concerning conceptual and practical learning. Section 2.3.1 thus focuses on research on conceptual learning, Section 2.3.2 discusses research on the role of practise in learning, while Section 2.3.3 reports research that relates students' learning of concepts and practise.

2.3.1 Learning the concepts

Higher education and educational research have long had an emphasis on concepts and conceptual learning, and consequently there exists a huge body of research in this area. In the following I will discuss research on conceptual learning with a focus on phenomenography, which is my main research approach, and some research from the conceptual change research tradition.

Phenomenography

There are a number of phenomenographic studies reported in computer science where the focus is on students' understanding of concepts. In 1992 Shirley Booth published her influential thesis "Learning to Program. A phenomenographic perspective" where the main research question was "What does it mean and what does it take to learn to program?". Examples of other phenomenographic work related to programming education are Berglund (2005), where the author discusses senior computer science students' learning of computer systems in a distributed project course. Eckerdal (2006) studied novice programming students' understanding of some central object-oriented concepts, and their use

of resources, while Boustedt (2007) studied senior computer science students working with a large software system, and how the students in this situation understand some advanced object-oriented concepts. Examples of studies that take the computer science teachers' perspective are Carbone et al. (2007), where teachers' understanding of successful and unsuccessful teaching is investigated, and Pears et al. (2007) where the focus is on teachers' experiences of students in trouble and the course content that troubles them.

Conceptual change
Molander et al. (2001) discuss that "a central problem has been to account for the conditions underlying the process of *conceptual change*" (p. 115). According to the authors, this change has "been regarded as an almost unquestioned goal for instruction." (p. 115) Molander et al. refer to Posner et al. (1982) , which they call an "influential article". Posner et al. discuss a theory of conceptual change:

> Learning is concerned with ideas, their structure and the evidence for them. [...] We believe it follows that learning, like inquiry, is best viewed as a process of conceptual change. The basic question concerns how students' conceptions change under the impact of new ideas and new evidence. (Posner et al., 1982, p. 212).

Davies and Mangan (2007) discuss conceptual change in economic education in relation to threshold concepts (Meyer and Land, 2005) and WTP: "[threshold concepts] might best be seen as a web of concepts which link thinking and practise in a discipline."

A recent discussion on conceptual change as it applies to learning and instruction is found in Vosniadou (2007), wherein Entwistle (2007) presents a broad review of research that relates to and has contributed to the conceptual change movement.

2.3.2 Learning the practise

Practise is generally accepted as the most important means to reach the learning goals in computer science education, and good practical skills are seen as the most important learning goal beside good conceptual knowledge.

Examples from computer science education research on the role of practise are studies of students' use of technology based resources like the computer, compilers, editors, and different kinds of software tools. There exists a considerable body of research on such resources in programming education (Valentine, 2004). This research includes discussions on choice of programming paradigm and language, the development of software tools, how students understand them, for example how students understand compiler messages and debugging, and how students use them. For an overview of this line of research, see Eckerdal (2006).

Gross and Powers (2005) discuss novice programmers' learning difficulties saying that teachers "have developed a myriad of tools to help novices learn to program. Unfortunately, too little is known about the educational impact of these environments."

Students' ability to write, read, and trace code is researched in some multi-national, multi-institutional studies: McCracken et al. (2001), Lister et al. (2004), and Lister et al. (2006). Students' ability to design is reported in a multi-national, multi-institutional study by Eckerdal et al. (2006). These papers all point to large deficiencies in students' practical skills.

2.3.3 Relating Thinking and Practising

There is little research on the relation between conceptual learning and practise within the subject area of computer science. In the area that Robins et al. (2003) call "psychological/educational study of programming", the complex relationship between conceptual learning and practise has however been recognized by du Boulay (1988) who discusses domains that programming students must learn to master. These include the syntax and semantics of a programming language and different programming skills. du Boulay writes:

> None of theses issues are entirely separable from each others, and much of the 'shock' [...] of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once. (du Bolay, 1988, p. 284).

Rogalski and Samurçay (1990) write:

> Acquiring and developing knowledge about programming is a highly complex process. [...] Even at the level of computer literacy, it requires construction of conceptual knowledge, and the structuring of basic operations (such as loops, conditional statements, etc.) into schemas and plans. (Rogalski and Samurçay, 1990, p. 170)

In addition to the above mentioned research from computer science, I have looked at educational research in science, mathematics, and technology where practical exercises and lab work play significant roles in the education, and where the role of practise, and the relation between practise and concepts, have been researched and debated. I will specifically discuss research from the conceptual/procedural knowledge research tradition and research on students' practical and conceptual learning in science lab since they seem close to what I study.

Conceptual versus procedural knowledge

In mathematics education research that has been inspired by cognitive psychology, there exists a considerable body of research where knowledge is largely divided into two types, conceptual and procedural, which have similarities with the distinction between Thinking and Practising made in the present research. This duality goes back to work by Hiebert and Lefevre (1986), but has been expanded over the years.

Baroody, Fail and Johnson (2007) give a brief overview of the work. They describe, with reference to Star (2005), the two knowledge types, where "each type of knowledge - procedural and conceptual - can either have a superficial or deep quality." (p. 115)

> he proposed defining *conceptual knowledge* as "knowledge of concepts or principles" – as knowledge that involves relations or connections (but not necessarily rich ones). He defined *procedural knowledge* as "knowledge of procedures" and *deep procedural knowledge* as involving "comprehension, flexibility, and critical judgment and [as] distinct from (but possibly related to) knowledge of concepts" (ibid., p 116, italics in original)

Baroody et al. (2007) further write with reference to Baroody (2003):

> although (relatively) superficial procedural and conceptual knowledge may exist (relatively) independently, (relatively) deep procedural knowledge cannot exist without (relatively) deep conceptual knowledge or vice verse (p. 123)

With reference to the same tradition, Hazzan (2003) discusses how the theory of process-object duality can be used to analyse students' understanding of computer science concepts.

Examples of research in technology education inspired by the same tradition is McCormick (1997). He writes: "technology education, in being concerned with both the practical and the intellectual, offers challenges to learning researchers." (p. 142) He claims that there is a need for researchers to "turn their attentions to the development of strategies for teaching problem solving and design" but "[m]ore complex still, they must show how conceptual knowledge relates to these procedures." (p. 155)

Practising and Thinking in the lab

Hofstein and Lunetta (2003) give a "critical review of the research on the school science laboratory", which follows up a similar study, (Hofstein and Lunetta, 1982). The authors discuss "the effectiveness and the role of laboratory work", that is not "as self-evident as it seemed" (p. 28). In their report from 1982 they write: "The research has failed to show simplistic relationships between experiences in the laboratory and student learning." (Hofstein and Lunetta, 1982, p. 212)

Séré (2002) reports on a project that intended "to address the problem of the effectiveness of lab work" in biology, physics, chemistry and mathematics. Research groups from seven European countries participated, and a number of case-studies were carried out at the upper secondary and undergraduate levels.

Hofstein and Lunetta, as well as Séré, discuss the importance of students' learning concepts and procedures through laboratory work, and both studies point to reported problems with laboratory activities: students do not necessarily learn concepts to the extent teachers expect and hope, and the procedures and technical details cause problems. Both studies also point to a lack of research in science education concerning the role of practise in laboratory work. Séré claims that this is less researched than conceptual learning in lab. Séré's report has its focus on the complex interaction between concepts and practise in laboratory work, and emphasizes the important role of procedures. She comments that this complexity "explains to a certain extent why conceptual learning

is not an automatic outcome of lab work." (p. 630) Hofstein and Lunetta have slightly different focus in their work than Séré. Their focus is on the importance of conceptual learning, that might be hidden when the main emphasis is on procedures and objects manipulated.

Séré's conclusion is that "To do" and "to learn to do" is as important as "to understand" and "to learn" (p. 638). Hofstein and Lunetta emphasize that teachers "need to be able to enable students to interact *intellectually* as well as *physically*, involving hands-on investigation and minds-on reflection" (p. 49, italics in original).

3 Research approaches

In order to better understand the complex relation between Thinking and Practising in programming education, I have re-analysed the empirical data from Study 1 and Study 2, using WTP as an overarching framework. In this section I will present my research approaches to explore Thinking and Practising.

3.1 Researching students' learning of Thinking

To research the role of Thinking in programming education I have focused on one particular aspect of Thinking, namely students' understanding of two central concepts, *object* and *class*. To this end, I have used phenomenography (Marton and Booth, 1997) and variation theory (Marton and Tsui, 2004). Phenomenography is a qualitative research approach which focuses on analysing and describing the variation in how people experience phenomena in the world[5]. A fundamental assumption in phenomenography is that for a given phenomenon there is a limited number of qualitatively different ways in which that phenomenon can be experienced in a certain situation.

Marton and Booth (1997) write about variation in peoples' capabilities for experiencing the world:

> These capabilities can, as a rule, be hierarchically ordered. Some capabilities can, from a point of view adopted in each case, be seen as more advanced, more complex, or more powerful than other capabilities. Differences between them are educationally critical differences, and changes between them we consider to be the most important kind of learning. (Marton and Booth, 1997, p. 111)

The result of a phenomenographic analysis is an outcome space, with a limited number of categories of description which show a "hierarchical structure of increasing complexity, inclusivity, or specificity" (ibid. p. 126). The categories describe the qualitatively different ways of experiencing the phenomenon

[5] A phenomenographic analysis can only reveal the researchers' interpretation of students' expressed *experiences* of a phenomenon. In the following text I will use *understanding* as interchangeable with *experience* since the present research discusses students' understandings of concepts.

that the researcher has identified in the data. Different categories reflect different combinations of features of the phenomenon which are present in the focal awareness at a particular point in time (ibid. p. 126). Learning is understood as to develop richer ways to see the phenomenon, as represented in the more advanced categories of the phenomenographic outcome space.

Variation and *discernment* are the key words in this process. According to variation theory, which originates from phenomenography, a necessary but not sufficient condition for discerning a specific feature of a phenomenon is that the student gets the opportunity to experience variation in a *dimension* corresponding to that feature. In Thuné and Eckerdal (2009) we explain dimensions of variation in the following way:

> For example, if 'size' and 'colour' are the features of a phenomenon 'picture component', then there is a 'size' dimension and a 'colour' dimension of the corresponding feature space. A particular instance of 'picture component' can be represented by its values in those dimensions, i.e. by its particular size and colour.

There is a possible dimension of variation for each feature of a phenomenon that can take different values. These values thus constitute a dimension, corresponding to the feature. In a learning context, some of these features and their relations are more critical to discern than others. A phenomenographic analysis can identify educationally critical features of the phenomenon studied.

According to variation theory, learning means developing richer ways of seeing a phenomenon, by becoming aware of additional features of it and of relations between the features. This, in turn, requires discerning the dimensions of variation corresponding to these additional features.

3.2 Researching elements of Practise in introductory programming courses

Practise is a wide term that can be used with many different meanings depending on the context. The focus of the present research is learning to *do* practise rather than learning *through* practise.

The students in Study 2 made a clear distinction between practical and non-practical learning goals, see Section 4.2. They discussed learning of programming concepts in terms of achieving *abstract (or theoretical) understanding*, *concrete understandings* (the ability to practise programming without necessarily understanding the underlying theories and concepts), the ability to *go from an understanding of the abstract concept to software design or concrete implementation*, an understanding of the *rationale* for learning and using the concept, and an understanding of how to *apply the concept to new problems*. I discuss practise in terms of the *concrete understanding*, the ability to *go from an understanding of the abstract concept to software design or concrete implementation*, and to be able to *apply concepts to new problems*.

I will further distinguish between *exercises*, *skills*, and *activities*. Exercises are here discussed in terms of practises where the students follow more or less

detailed instructions prepared by the teacher. Exercises are less discussed than the other two, since they represent learning *through* practise.

Skills are practical knowledge students are supposed to acquire during their education. Typical programming skills novice students are expected to learn are to read, to write, and to debug simple computer program code. Each skill is manifested in many different activities that the students are supposed to learn, and these activities demand different levels of proficiency to be properly performed. Skillfulness in programming requires long experience and good theoretical understanding.

For the investigation of Practising in the present study, I have analysed three skills that are often introduced early in the education: to read, to write, and to test and debug code. These skills constitute cornerstones in the implementing and testing phase of software development, see Section 2.1.

The analysis of the practise aims at producing detailed lists of activities that reflect the three skills with the focus on novice students. At the same time this analysis reveals the wide distribution of levels of proficiency inherent in each skill.

The list of activities have been identified in the data, but complemented with activities typically found in text books, explicitly expressed or tacitly presupposed. The reason to look for typical novice activities outside the data is twofold. First, neither of the two studies had an emphasis on learning activities, which limit the discussion on activities found in the data. Second, many activities in programming are tacit and implicit, a kind of taken-for-granted knowledge that is seldom explicitly expressed. The activities can however be identified by for example close examination of examples and figures in text books presenting object-oriented programming for novices.

4 Empirical findings

Before I present the results of the phenomenographic analysis and the analysis on novice programming activities, respectively, some empirical findings are presented to the reader. The following quotes from interviews with students are organised around two identified themes that were salient in the data. Each theme is discussed from a few different aspects that emerged from the data and illuminate the themes.

The first theme, salient in both studies but primarily in Study 1, is that practise plays an important but problematic role in programming students' learning. The data gives evidence that learning of the practise is as important to research as learning of the concepts. This is discussed in Section 4.1.

The second theme is the complex relation between practise and conceptual learning in the students' learning process, which was specifically apparent in Study 2. This is discussed in Section 4.2.

In this way this section provides empirical evidence that students' learning of concepts and practise need to be researched simultaneously. Consequently the subsequent Section 5 demonstrates a way to research this relation by combining

the phenomenographic results and the results of the analysis of common novice students' activities.

The students from Study 1 will in the following section be referred to as Student A, Student B, Student C (A, B, C) etc, with no reference to their real names, while the students in Study 2 will be referred to as Student 1, Student 2, Student 3 (S1, S2, S3) etc.

4.1 The important but problematic role of Practise in programming education

Three aspects of this theme will be discussed. First we establish that practise is not merely a tool to reach the conceptual learning goals, but is part of the learning goals. Second, practise, in terms of exercises, is the generally accepted, and often presupposed, main road to learning programming in computer science education. Third, data from Study 1 and Study 2 indicate that students do not learn through the exercises to the extent they are expected to. This applies both to the conceptual and the practical learning.

First aspect: Practise as part of the learning goals
There is agreement in industry and among educators that profound practical knowledge is a central goal in programming education, and that this often requires many years of hands-on training. The novice students in Study 1 and the senior students in Study 2 are well aware of the important role of the practise.

The students in Study 1 were asked about what learning to program means. All students expressed that learning to program is experienced as learning to understand some programming language, and using it for writing code. This emphasises the practical use of a programming language. In addition, Student D emphasises the ability to read, trace and debug code:

> D:[...] to see a program and see that, okay, this will happen and this is what the computer will do, this will be performed. And then also to see what's wrong in the language, to discover errors when you program and to see that this will not work because this can't be written like that.

Second aspect: Practise as means to reach the learning goals
Practise as a way to learn is emphasised by students in both studies. Student 7 in Study 2 emphasises the importance of learning step-by-step procedures, or "templates" for doing things. This "de-mystifies" the concept and helps the student to reach a better understanding. The student discusses pointers which is generally regarded as a hard concept to learn, even a threshold concept:

> S7: [...] when I got the idea of this format, this template for creating nodes and node datas and how they can go together and kind of a step by step then it de-mystifies it actually. I can see if I missed a step. "Oh, dummy, you forgot to hook these two together," and maybe that's what in a lot of programming with practise you mentally get these steps in your mind.

Third aspect: Problems with the practise hinder further learning
The third aspect, salient in the data from both Study 1 and Study 2, highlights the problematic role of practise in students' learning. If the students have problems with learning *through* practise, conceptual learning as well as learning to do practise might be hindered. A few examples of this are shown below.

Student D in Study 1 answers the question what has been difficult in the course. In the quote below he or she expresses problems in learning concepts as well as in mastering the practise, "to understand [...] how one should use different things in a program" and "how to study, when you implement the programs".

> D: Yes, I think it has been difficult with concepts and stuff, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult [...] But I still think the course, it's difficult for a novice to sort of get a grip of how to study, when you implement the programs and like that.

Student D further discusses the contrast between learning mathematics and learning computer programming:

> D: I've taken many math courses but math is kind of logical and you understand it but this is... no I don't know. [...] Here you feel as if you only learn a lot of examples. You know, we've gotten so many examples of everything, in some way it feels as if you don't understand the basis from the beginning [...]

The examples student D talks about are provided by the teacher and intend to show some of the basic practises that are needed in programming: how to structure a program, how some data structures are implemented and used in algorithms through control structures etc. Still the student has not discerned the practise sufficiently to know "how to study", as he or she puts it in the quote above.

There was evidence also in Study 2 that the practise can be a major obstacle in learning to program. Student 7 in Study 2 emphasises that it is the practise that is the problem in the learning:

> S7: There's just some aspects to it that just seem to remain kind of mysterious to me at the programming level. Not the concept level, not the theory level, not the technology level but at the kind of code nuts and bolts level.

Practise is, according to many students, the main road or doorway into further learning in programming. But learning through programming exercises, implies an enormous theoretical, practical and technological challenge for many novice students. In fact, the first threshold novice students often encounter involves a variety of different sorts of practise, and without knowledge of how to master the practise, there might be no way into further learning, neither conceptual nor practical. Practise is not merely the unproblematic road to the more advanced conceptual understanding.

4.2 The complex relation between Thinking and Practising in programming students' learning

Practise plays an important and often inevitable role in conceptual learning. This section discusses the second theme found in the data which deals with the complex relation between practise and conceptual learning in students' learning to program. Three aspects of the theme are discussed. The first aspect points to the close relation between concepts and practise in the learning which makes the novices discuss learning to program as learning a new way of thinking. The second aspect highlights the senior students' descriptions where distinct "parts" of the learning process have been identified. The third aspect shows that there is not one linear route through the learning process. There are rather individual routes, where different students get stuck at different places.

First aspect: The magic "Programming Thinking"
In both studies there was evidence that novice students found learning to program hard. Some students even experienced programming as "magic", or even frightening, in the beginning of their learning.

Strikingly many novice students in Study 1 discussed learning to program in terms of learning a special way of thinking. Inspired by the data, we introduced the term *programming thinking* to describe this phenomenon, different from, and for many novice students with troubling little coherence with, thinking in other subjects with which they had previous experience (Eckerdal and Berglund, 2005).

A quote from Student D in Study 1, which was cited in Section 4.1, is relevant also in this context. The student talks about his or her problem to learn to program:

> D: Yes, I think it has been difficult with concepts and stuff, as to understand how to use different, how one should use different things in a program. And I actually think that most of it has been difficult, but this very thought behind, it feels as some people just understand programming

Student A expresses this when answering the question what is most important in the course:

> A: It's more the thinking itself, the logical thinking. Everything you need to know you must think of when it comes to programming. [...] you've kind of got a small insight into what it's like to program and how the computer works like that, or the software.

Student D connects the problems associated with this way of thinking with conceptual learning as well as the practise "how one should use different things in a program". Student A moreover points to the problem of understanding the hardware, the computer itself, "how the computer works".

The novice students' descriptions of "programming thinking" illustrate how learning to program involves both conceptual learning and practise. Furthermore they show how closely related, and how dependent on each other the two

aspects are. The rest of this section will focus on the senior students in Study 2.

Second aspect: Parts of the conceptual and practical learning space
As the novices, the senior students discussed the complex relation between concepts and practise when learning to program, but unlike the novices the senior students could articulate such experiences in detail. For many novice students, the learning experience was "magic" and fragmented, while many senior students explicitly described crucial "parts" of the learning process.

Eckerdal et al. (2007) analysed the data from Study 2 in order to identify specific characteristics for computer science students who are in the midst of learning a threshold concept (Meyer and Land, 2005). The analysis was inspired by Meyer and Land's description of the *liminal space* which is a space in which someone is transformed, acquires new knowledge, and acquires a new status and identity within a community. We found that the students discussed the learning process of threshold concepts in terms of acquiring different distinct parts of a full understanding. The different partial understandings found are:

- abstract (or theoretical) understanding
- concrete understanding (the ability to practise programming without necessarily understanding the underlying concepts)
- the ability to go from an understanding of the abstract concept to software design or concrete implementation
- an understanding of the rationale for learning and using the concept
- and an understanding of how to apply the concept to new problems.

The senior students clearly pointed to theoretical as well as practical learning goals.

A few quotes from the students will illustrate some of the understandings expressed above.

Student 8 expresses the third understanding, the need of both an abstract understanding of a concept and the ability to relate it to implementation, which requires practise:

> S8: [...]the abstract understanding is something you learn by education, by reading, you can learn that in class, but the understanding of actually applying it to programs you can't, you must, you must learn it by, by, by using it

The last understanding in the list above, "an understanding of how to apply the concept to new problems", mirrors that students distinguish relating abstract concepts to implementation of routine problems, and applying concepts to *new* problems. The latter is experienced as specifically problematic for some students. This is expressed by Student 2:

> S2: You understand how a theory works but how do you take that theory and how it works and apply it to a practical sense? I think that is one of the hardest leaps to make.

Third aspect: Individual routes in a practical and theoretical learning space

In the analysis of students in the liminal space (Eckerdal et al., 2007) we found that the different "parts" of the learning process were discussed by the students as inseparable facets of one and the same learning experience. Different students struggled with different parts, depending on, for example, background knowledge. Some students expressed that they first learned the theory behind the concepts, and then how to use them, while other students expressed the opposite. Some discussed that the major problem was to understand the rationale behind some concepts, while other students did not mention that aspect at all. The observed differences in the students' learning processes need however to be further researched before we can draw any deeper inferences from them.

The data clearly indicated that there is no common route for the students through the learning space. Rather a complex pattern of individual learning routes appeared. Still, both concepts and practise seem important, and both can cause problem in the learning process.

This section has discussed some salient themes in the empirical data: practise and concepts are both parts of the learning goals and dependent of each other in the learning process. They mutually support each other in the learning, but at the same time either of them can become an obstacle for further learning. They are complexly interwoven in the learning process, and play different roles depending on the individual. For some novice students this was experienced as an alien and magic "programming thinking" which was difficult to grasp. The senior students could articulate a complex learning space where both concepts and practise played important roles. In the following section, the data is reanalysed to get a better understanding of how Thinking and Practising relate in students' learning of computer programming.

5 Ways of Thinking and Practising - an analytic model

I will now continue with an in-depth analysis of WTPs from the novice programming students' perspective using data mainly from Study 1. The aim is to analyse how Thinking, here discussed as students' understanding of concepts, and Practising, here discussed as common novice students' programming activities, are related. First however, Thinking and Practising are discussed separately. Based on these two discussions, I will then demonstrate how variation theory can be used to analyse and display the relation between Thinking and Practising in programming students' learning process.

5.1 Thinking manifested in students' understanding of concepts

To investigate Thinking in the form of conceptual understanding, we made a phenomenographic analysis of novice students' understanding of the central concepts *object* and *class*, based on data from Study 1. The two concepts object and class are closely related, and the analysis resulted in one phenomenographic outcome space, see Table 1. The categories in the outcome space are descriptions of the qualitatively different ways in which the concepts object and class are seen by the group of interviewees on a collective level. These categories are inclusive which means that the understandings described in the latter categories include the understandings described in the former. The latter categories thus reflect richer ways to understand the concepts.

Having formulated the phenomenographic outcome space summarised in Table 1, we made a subsequent analysis to identify dimensions of variation related to the two concepts considered here. As discussed in Section 3.1, possible values of a certain feature of a phenomenon constitute a dimension of variation for this phenomenon. In order to identify dimensions of variation we consequently re-examined the categories of description in Table 1, to find what features of the two concepts that are in focus in the different understandings expressed. For a more comprehensive presentation of the phenomenographic analysis where the dimensions of variation are identified, see Eckerdal and Thuné (2005) and Eckerdal (2006).

Object is experienced as a piece of program text, and class as an entity of the program that structures the code and describes the object.
Object and class are experienced as expressed in the previous category. In addition, class is experienced as a description of properties and behavior of objects, and object as something that is active when the program is executed.
Object and class are experienced as expressed in the previous category. In addition, class is experienced as a description of properties and behavior of objects, where the object is a model of some real world phenomenon.

Table 1: A phenomenographic outcome space on novice students' understanding of the concepts *object* and *class*. For details, see Eckerdal and Thuné (2005) and Eckerdal (2006).

Below I discuss and label the different dimensions of variation identified.

In the first category of description in Table 1, the feature in focus is the textual representation of the concepts. I will refer to the corresponding dimension

of variation as TEXT.

In the second category, the new feature added is the active behaviour when the program is executed. I will refer to the corresponding dimension of variation as ACTION.

The new feature described in the third category is the modeling aspects of the concepts, and I will refer to corresponding dimension of variation as MODEL.

5.2 Practising manifested in common students' activities

This section aims at analysing the role of practise in novice programming students' learning. The analysis of the practise is not a traditional phenomenographic analysis, but an analysis based on elements from variation theory on common and important novice students' programming activities.

In Study 1 and Study 2 we found students discussing different skills that are important when learning to program. Examples of such skills, typical for novice programming are to write code, read code, test code, debug code, design software, and use advanced technical resources, for example different IDEs.

In this section I will focus on a few of these skills namely to read, to write, and to test and debug code. To test and debug code will be treated as one skill since they are closely connected. The skills focused on in the present discussion are often seen as the most fundamental skills in programming and essential for novice programming students to learn.

The skills are manifested in several more or less advanced activities. Aiming at relating the activities and students' understanding of concepts, I will first list such activities that are important and frequently occurring in novice programming courses, see Table 2. In addition to the activities mentioned by the students in the data from Study 1 and Study 2, activities commonly found in text books are added.

Although some activities do not cause problems for most students, they are still included to demonstrate the breadth of new activities and tools novice students meet and are expected to learn and use at an early stage of their education. The detailed list is in line with suggestions from science education research, where Séré writes: "A first step in research should now be to describe what happens during lab work as exhaustively as possible." (Séré, 2002, p. 628). A detailed list concretises what practise means in novice students' programming. The list above covers common novice programming activities reasonably well for the present discussion.

Why do some students have such big problems in performing some activities? Activities carry meaning and it is important that students discern this meaning. Runesson (2006), with reference to Svensson (1984), discusses meaning in relation to learning:

> [meaning] is constituted as a relation between the object to which I direct my awareness and me, the subject. The meaning emerges as I direct my awareness to the object. (Runesson, 2006, p. 400)

Read code: to discern main parts of short programs; to read code and recognize key words; to read code and understand what will happen when the instructions are executed; to read and relate code to the application and the problem domain.
Write code: to use an editor to emphasis the structure of a program by means of indents, empty lines etc.; to write common programming building blocks in a syntactically correct way; to design a short algorithm; to express a short algorithm in pseudo code; to implement pseudo code in a programming language; to design a solution to a whole problem and transfer the design to pseudo code, using common programming building blocks; to implement the solution to a problem according to basic software quality requirements.
Test and debug code: to use a compiler to find and correct minor syntax errors; to use the computer to execute code to verify expected output; to use a compiler to get executable code; to read and understand error messages about simple syntax errors, such as missing semicolon; to correct simple syntax errors, for example missing semicolon; to hand execute a program on paper before coding; to diagnose semantic errors in the code; to test code in relation to the problem domain and usability.

Table 2: Common novice programming skills with associated activities.

Runesson (2006) furthermore explains the relation between meaning and dimensions of variation:

> the meaning we assign something is constituted as a pattern of simultaneously discerned dimensions of variation. (Runesson, 2006, p. 403)

In my interpretation of practise, an activity is the object towards which the awareness is directed. To discern the meaning of a certain activity the student thus has to simultaneously be focally aware of certain dimensions of variation related to this activity.

To elaborate on this argument I will discuss the activities that are related to the skill *read*, and leave to the reader to extrapolate the discussion to the activities related to the other skills. The activities "to discern main parts in short programs", and "to read code and recognize key words" relate to the text-representation of the code. To discern the meaning of these activities, the students need to become aware of the TEXT dimension of variation discussed in the previous section. On the other hand, in order to master the activity "to read code and understand what will happen when the instructions are executed" the students need to become aware of the ACTION dimension of variation in addition to the TEXT dimension. Finally, the activity "to read and relate code to the application and the problem domain" requires that the students discern not only the TEXT and the ACTION dimensions of variation, but also the

MODEL dimension. It is necessary in this discussion to say that there might be other dimensions of variation related to the activities mentioned that have not come to the foreground in this analysis, but I claim that the students need to discern at least these dimensions.

Activities related to the TEXT dimension of variation discern main parts of short programs; read code and recognize key words; use an editor to emphasize the structured of a program by means of indents, empty lines etc.; use the computer to execute code to verify expected output; write common programming building blocks in a syntactically correct way; use a compiler to find and correct minor syntax errors; use a compiler to get executable code
Activities related to the TEXT and ACTION dimensions of variation read code and understand what will happen when the instructions are executed; design a short algorithm; express a short algorithm in pseudo code; implement pseudo code in a programming language; hand execute a program on paper before coding; diagnose semantic errors in the code
Activities related to the TEXT, ACTION, and MODEL dimensions of variation read and relate code to the application and the problem domain; test code in relation to the problem domain and usability; design a solution to a whole problem and transfer the design to pseudo code, using common programming building blocks; implement code according to basic software quality requirements

Table 3: Common novice programming activities at different levels of proficiency. The activities can be experienced as meaningful when related dimensions of variation are discerned.

This discussion can contribute to explain students' problems with reading code as reported by Lister et al. (2006). They conclude that students need to be able to for example manually trace code, which requires that the TEXT and the ACTION dimensions have been discerned, according to the above reasoning. This is however not sufficient "if they are to develop as programmers." (ibid. p. 122). The students furthermore need to be able to "read several lines of code and integrate them into a coherent structure - to see the forest, not just the trees." (ibid. p. 122). The activity described in this quote seem to require that the MODEL dimension, as well as the ACTION and TEXT dimensions have been discerned, and many novice students have problems reaching this level of proficiency. Similar discussions can be made for all activities in Table 2.

In Table 3 the activities are re-grouped according to different combinations of dimensions of variation. Students can become aware of the meaning embedded in an activity if the dimensions of variation related to the activity are discerned. By studying Table 3 it becomes visible that activities that relate to

more dimensions of variation correspond to *a higher level of proficiency* than the activities that relate to fewer dimensions.

5.3 Developing an analytical model for relating Thinking and Practising

The discussions in Section 5.1 and Section 5.2 show that both conceptual understandings and activities are related to dimensions of variation. There are qualitatively more advanced ways to understand the concepts that relate to more dimensions of variation. In similar ways there are activities at higher level of proficiency which relate to more dimensions of variation. Table 4 merges Table 1 and Table 3 and illustrates that the dimensions of variation are in the center of the analysis, relating conceptual understandings and activities to each other.

The structure of Table 1 and Table 3 is reflected in Table 4. The left column includes the categories of the phenomenographic outcome space from Table 1. The activities listed in Table 3 are found in the right column. In the middle column are the dimensions of variation, related to the qualitatively different conceptual understandings as well as to the activities at different levels of proficiency.

The relations described in Table 4 between the activities at different levels of proficiency and the different understandings of the concepts *object* and *class* are examples of how Practise and Thinking are related, and a different example could have been chosen to make my point. The significant implication is that *both* to discern a certain feature of a concept and to make an activity meaningful require that certain dimensions of variation in the learning space is opened for the student. It is the dimensions of variation that are at the center of this discussion.

The empirical findings indicate that when students have reached a certain level of practical proficiency, this can help them in their learning of new concepts, and that understanding of concepts can help them to master new practise, see Section 4.2. These empirical findings are in line with Table 4 where the dimensions of variation are at the center. The learning of concepts *and* activities presupposes that related dimensions are discerned. Once a dimension of variation is discerned, this can help the students to understand related concepts (Marton and Tsui, 2004) *and* to learn related activities.

Table 4 illustrates my analytical model where the dimensions of variation are at the center of the discussion on students' learning. Figure 1 is a more abstract and general illustration of the model. It shows that several activities as well as several concepts can be related to a certain dimension of variation. This finding is explicitly shown in Table 4. Moreover, Figure 1 shows that activities as well as concepts can be related to more than one dimension of variation.

An example of this is that we found both the TEXT and the ACTION dimensions in the phenomenographic analysis of the students' experiences of what computer programming means (Thuné and Eckerdal, 2009). These dimensions are also found in the present analysis of the same students' understanding of the

Understanding class and object as	Dimensions of variation	Typically novice students' activities that are examples of the skills read, write, and test and debug code
Object: piece of program text. Class: entity of the program that structures code and describes the object.	TEXT	- discern main parts of short programs - read code and recognize key words - use an editor to emphasize the structure of a program by means of indents, empty lines etc. - execute code to verify expected output - write common programming building blocks in a syntactically correct way - use a compiler: find and correct minor syntax errors - use a compiler to get executable code
In addition, object: active during execution. Class: describe properties and behaviour.	TEXT and ACTION	- read code and understand what will happen when the instructions are executed - design a short algorithm - express a short algorithm in pseudo code - implement pseudo code in programming language - hand execute program on paper before coding - diagnose semantic errors in the code
In addition, object: model real world phenomenon. Class: describes properties and behaviour	TEXT, ACTION, and MODEL	- read and relate code to the application and the problem domain - test code in relation to the problem domain and usability - design a solution to a whole problem and transfer the design to pseudo code, using common programming building blocks - implement code according to basic software quality requirements

Table 4: Categories describing students' understanding of the concepts object and class (left column), and novice students' activities at different levels of proficiency (right column) are related to dimensions of variation (middle column).

concepts object and class, and the dimensions are further related to programming activities at certain level of proficiency. An inference of this finding is that the student can discern for example the ACTION dimension either through a related activity (for example one of those mentioned in Table 4), or when studying the concepts object and class, or when learning what computer programming means, or through learning other concepts or activities. Furthermore, once the student has discerned the dimension, the discernment can help the student to learn the other concepts and activities that are related to this dimension.

The present work has identified that practise as well as concepts have related dimensions of variation. Fazey and Marton (2002) discuss dimensions of variation related to practise. The authors summarise a number of studies on systematic variation of practising motor skills saying:

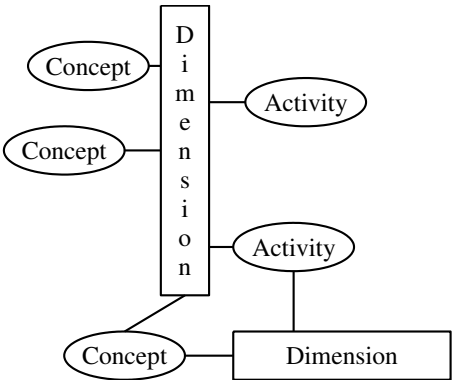

Figure 1: The figure illustrates two dimensions of variation of a learning space. Empirical findings indicate that the dimensions can act as interfaces between different conceptual understandings and activities. When a dimension is discerned, this can open for learning concepts *and* practises in new ways.

> What comes out of these examples for us is not simply that varying practise conditions can have positive, longer term effects for learners, but also that there are several dimensions along which practise conditions might vary. Some of these are embedded [...] in the way learners are exposed to variations in practise. (ibid. p. 244).

The new significant finding in the present research is however that practise and conceptual understandings are related through *common* dimensions of variation.

Table 4 indicates that depending on the dimensions of variation discerned by the student, it is likely that the student can learn activities (practises) related to only those dimensions, since it is then possible for the student to see the meaning of the activities. In a similar way, some ways to understand the concepts are possible for the student to discern. The higher level of practical proficiency and the more advanced level of conceptual understanding, the more features, or the more dimensions of variation and their relations, the student needs to discern.

In conclusion Figure 1 demonstrates the complex relation between Thinking and Practising in the learning process, where individual students take different routes. Students can discern a dimension of variation when studying concepts *or* working with the activities. When a dimension is discerned, this opens the possibility for a wider conceptual understanding *and* for learning related activities. This is in line with the finding that some students express that they have discerned the activities first, and then the concepts, while other students express the opposite learning experience.

6 Conclusions and future work

The research presented in this paper demonstrates the complex relationships and mutual dependence between novice programming students' conceptual and practical learning. I have used Ways of Thinking and Practising, WTP, as a conceptual framework in the investigation of this relationship. In the present work Thinking refers students' conceptual understandings, while Practising refers to common novice students' programming activities. In this context the relation between Thinking and Practising means how conceptual learning and learning of programming activities depend on each other and mutually carry meaning to each other and make learning possible or hindered.

Empirically the research builds on two interview studies with computer science students. For the analysis of the data I have primarily used phenomenography and variation theory.

The main findings are the following:

- The practise is experienced by many students as equally, or even more troublesome to learn than the concepts. Furthermore it is shown that students experience programming activities and conceptual understanding as equally important. If they face problems with one of them, they are likely to face problems with the other.
- An analytical model of students' learning is proposed that demonstrates that activities as well as conceptual understandings relate to dimensions of variation.
- Higher level of practical proficiency relate to more dimensions of variation in a similar way as more advanced ways to understanding concepts relate to more dimensions of variation.
- The most significant finding in the present research is that I have demonstrated that practise and conceptual understandings have dimensions of variation *in common*. This has been possible since I have showed a way to identify dimensions of variation related to practises.
- Consequently, when a dimension of variation is opened for a student, this creates an opportunity for the student to learn concepts *and* activities in new ways.

These results can explain how students' learning sometimes seems to go from concepts to practise, and sometimes from practise to concepts. In addition this explains that when a certain concept *or* activity is learned, it can open up for learning new concepts and activities, related to those already learned, via the dimensions of variation.

Moreover, the results of the present research can to some extent explain why activities, performed for example in the lab, do not necessarily lead to deepened conceptual understanding, and why studying of concepts do not necessarily lead

to a higher level of skillfulness in programming education. If the student in the learning situation does not discern the related dimensions of variation, there might be no corresponding conceptual and practical learning.

Phenomenography and variation theory (Marton and Booth, 1997; Marton and Tsui, 2004) traditionally discuss ways to identify critical features of concepts, and ways to open a space of learning for students by means of patterns of variation in the teaching. The present work contributes to the body of knowledge of students' learning by proposing an analytical model of how dimensions of variation relate to conceptual and practical learning. To use the model as a basis for further empirical studies about the relation between practise and concepts in a learning context would be an interesting line of future research.

Acknowledgment

I want to thank my supervisor Professor Michael Thuné for his support during the research process; in rewarding discussions and for constructive ideas, and for his patience and help during the process of writing the paper.

References

Baroody, A. (2003). The development of adaptive expertise and flexibility: The integration of conceptual and procedural knowledge. In Baroody, A. and Dowker, A., editors, *The development of arithmetic concepts and skills: Constructing adaptive expertise*, pages 1–34. Erlbaum, Mahwah, NJ.

Baroody, A., Feil, Y., and Johnson, A. (2007). An alternative reconceptualization of procedural and conceptual knowledge. *Journal for Research in Mathematics Education*, 38(2):115–131.

Berglund, A. (2005). *Learning computer systems in a distributed project course. The what, why, how and where*. Number 62 in Uppsala Dissertations from the Faculty of Science and Technology. Acta Universitatis Upsaliensis, Uppsala, Sweden.

Booth, S. A. (1992). *Learning to Program. A phenomenographic perspective*. Number 89 in Göteborg Studies in Educational Science. Acta Universitatis Gothoburgensis, Göteborg, Sweden.

Boustedt, J. (2007). *Students Working with a Large Software System: Experiences and Understandings*. Licentiate thesis, Uppsala University, Uppsala, Sweden.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? *SIGCSE Bulletin*, 39(1):504–508.

Carbone, A., Mannila, L., and Fitzgerald, S. (2007). Computer science and it teachers' conceptions of successful and unsuccessful teaching: A phenomenographic study. *Computer Science Education*, 17(4):275–299.

Davies, P. and Mangan, J. (2007). Threshold concepts and the integration of understanding in economics. *Studies in Higher Education*, 32(6):711–726.

du Bolay, B. (1988). Some difficulties of learning to program. In Soloway, E. and Spohrer, J., editors, *Studying the Novice programmer*, pages 283–299. Lawrence Erlbaum Associaties Inc.

Eckerdal, A. (2006). *Novice Students' Learning of Object-Orientd Programming*. Licentiate thesis, Uppsala University, Uppsala, Sweden.

Eckerdal, A. and Berglund, A. (2005). What Does It Take to Learn 'Programming Thinking'? In *Proceedings of the 1st International Computing Education Research Workshop, ICER*, pages 135–143, Seattle, Washington, USA.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliff, M., and Zander, C. (2006). Categorizing student software designs: Methods, results, and implications. *Computer Science Education*, 16(3).

Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., Thomas, L., and Zander, C. (2007). From Limen to Lumen: Computing students in liminal spaces. In *Proceedings of the 3rd International Workshop on Computing Education Research*, pages 123–132. ACM.

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

Entwistle, N. (2003). Concepts and conceptual frameworks underpinning the ETL project. Occasional Report 3 of the Enhancing Teaching-Learning Environments in Undergraduate Courses Project, School of Education, University of Edinburgh, March 2003.

Entwistle, N. (2007). Conceptions of learning and the experience of understanding: Thresholds, contextual influences, and knowledge objects. In Vosniadou, S., Baltas, A., and Vamvakoussi, X., editors, *Re-framing the Conceptual Change Approach in Learning and Instruction*, pages 123–143. ELSEVIER.

Fazey, J. and Marton, F. (2002). Understanding the space of experiential variation. *Active Learning in Higher Education*, 3(3):234–250.

Gross, P. and Powers, K. (2005). Evaluating assessments of novice programming environments. In *Proceedings of the 1st International Computing Education Research Workshop, ICER, Seattle, Washington, USA*, pages 99–110.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of computer science. *Computer Science Education*, 13(2):95–122.

Hiebert, J. and Lefevre, P. (1986). Conceptual and procedural knowledge in mathematics: An introductory analysis. In Hiebert, J., editor, *Conceptual and procedural knowledge: The case of mathematics*, pages 1–27. Erlbaum, Hillsdale, NJ.

Hofstein, A. and Lunetta, V. (1982). The role of the laboratory in science teaching: Neglected aspects of research. *Review of Educational Research*, 52(2):201–217.

Hofstein, A. and Lunetta, V. (2003). The laboratory in science education: Foundations for the twenty-first centry. *Science Education*, 88(1):28–54.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4):119–150.

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 *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 146–165, New York, NY, USA. ACM.

Marton, F. and Booth, S. (1997). *Learning and Awareness*. Lawrence Erlbaum Ass., Mahwah, NJ.

Marton, F. and Tsui, A. (2004). *Classroom Discourse and the Space of Learning*. Lawrence Erlbaum Ass., Mahwah, NJ.

McCartney, R., Eckerdal, A., Mostrom, J. E., Sanders, K., and Zander, C. (2007). Successful students' strategies for getting unstuck. *SIGCSE Bulletin*, 39(3):156–160.

McCormick, R. (1997). Conceptual and procedural knowledge. *International Journal of Technology and Design Education*, 7(1–2):141–159.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.-D., 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. *SIGCSE Bulletin*, 33(4):125–180.

McCune, V. and Hounsell, D. (2005). The development of students' ways of thinking and practising in three final-year biology courses. *Higher Education*, 49:255–289.

Meyer, J. H. and Land, R. (2005). Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):373–388.

Molander, B. (1996). *Kunskap i handling*. DAIDALOS.

Molander, B., Halldén, O., and Pedersen, S. (2001). Understanding a Phenomenon in Two Domains as a Result of Contextualization. *Scandinavian Journal of Educational Research*, 45(2):115–123.

Pears, A., Berglund, A., Eckerdal, A., East, P., Kinnunen, P., Malmi, L., McCartney, R., Moström, J., Murphy, L., Ratcliffe, M., Schulte, C., Simon, B., Stamouli, I., and Thomas, L. (2008). What's the problem? teachers' experience of student learning successes and failures. *Proc. 7th Baltic Sea Conference on Computing Education Research: Koli Calling, CRPIT, Australian Computer Society*, 88.

Posner, G., Strike, K., Hewson, P., and Gertzog, W. (1982). Accommodation of a scientific conception: toward a theory of conceptual change. *Science Education*, 66(2):211–227.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

Rogalski, J. and Samurçay, R. (1990). Acquisition of programming knowledge and skills. In Hoc, J., Green, T., Samurçay, R., and Gillmore, D., editors, *Psychology of programming*, pages 157–174. Academic Press.

Runesson, U. (2006). What is it Possible to Learn? On Variation as a Necessary Condition for Learning. *Scandinavian Journal of Educational Research*, 50(4):397–410.

Star, J. R. (2005). Reconceptualizing procedural knowledge. *Journal for Research in Mathematics Education*, 36:401–411.

Svensson, L. (1984). Människobilden i INOM-gruppens forskning: Den lärande människan. [The image of man in research in the INOM-group; in Swedish]. Technical report, Göteborgs universitet, Institutionen för pedagogik, Sweden.

Séré, M. (2002). Towards renewed research questions from the outcomes of the european project *Labwork in Science Education*. *Science Education*, 86(5):624–644.

Thuné, M. and Eckerdal, A. (2009). Variation Theory Applied to Students' Conceptions of Computer Programming. *European Journal of Engineering Education*, Accepted for publication.

Valentine, D. (2004). CS educational research: a meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE Bulletin*, 36(1):255–259.

Vosniadou, S., Baltas, A., and Vamvakoussi, X. (2007). *Reframing the Conceptual Change Approach in Learning and Instruction (Advances in Learning and Instruction)*. Elsevier.