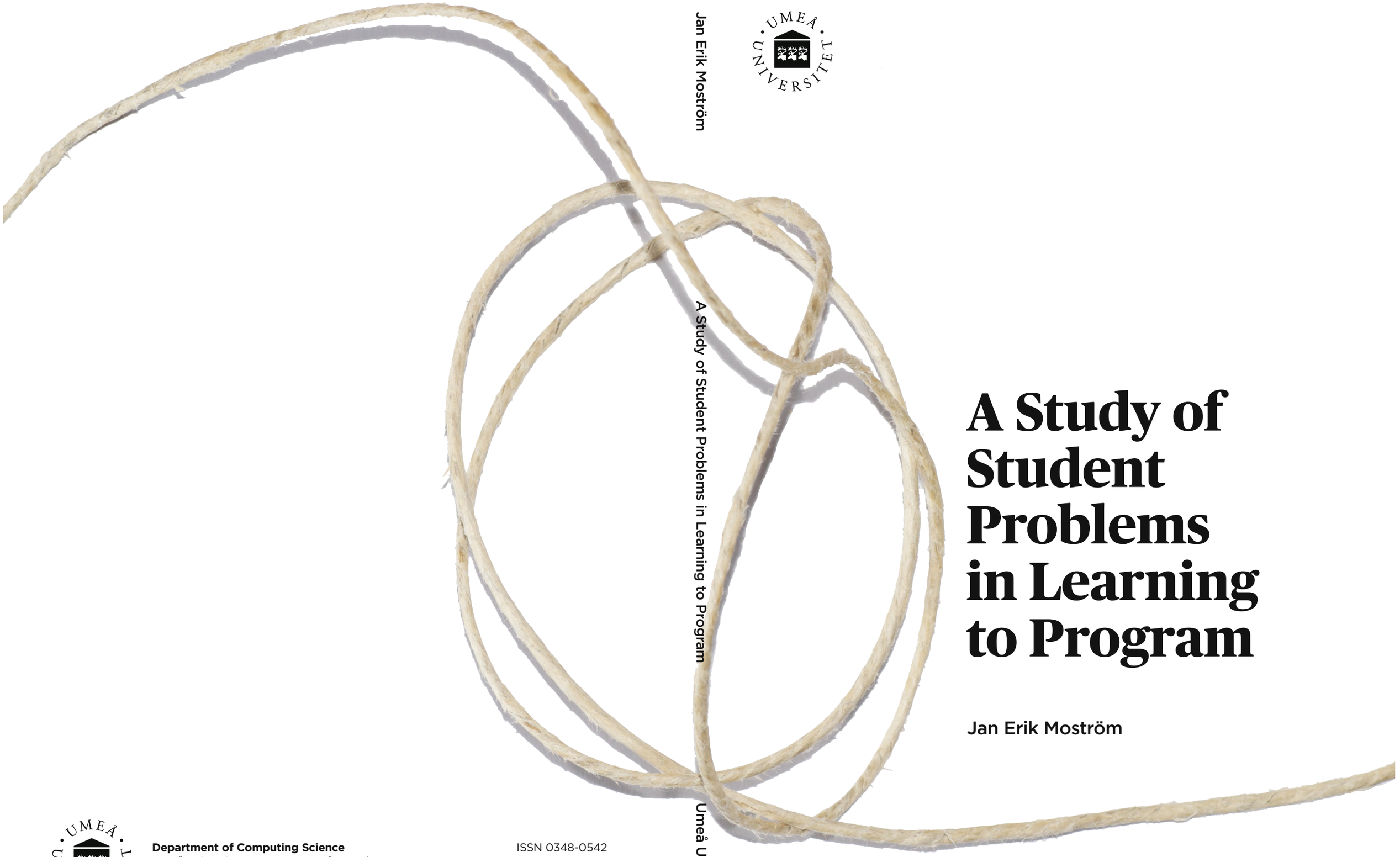

A Study of Student Problems in Learning to Program

Jan Erik Moström

Department of Computing Science
Umeå 2011

A Study of Student Problems in Learning to Program
Jan Erik Moström
Umeå University

Department of Computing Science
Umeå university, SE-901 87 Umeå, Sweden
www.cs.umu.se

ISSN 0348-0542
ISBN 978-91-7459-293-1
UMINF 11.10

DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

jem@cs.umu.se

ISBN 978-91-7459-293-1
ISSN 0348-0542
UMINF 11.10

Printed by Print & Media, Umeå University, 2011

Abstract

Programming is a core subject within Computer Science curricula and many also consider it a particularly difficult subject to learn. There have been many studies and suggestions on what causes these difficulties and what can be done to improve the situation.

This thesis builds on previous work, trying to understand what difficulties students have when learning to program. The included papers cover several areas encountered when trying to learn programming.

In Paper I we study how students use annotations during problem solving. The results show that students who annotate more also tend to be more successful. However, the results also indicate that there might be a cultural bias towards the use of annotations.

Not only do students have problems with programming, they also have problems with designing software. Even graduating students fail to a large extent on simple design tasks. Our results in Paper II show that the majority of the students do not go beyond restating the problem when asked to design a system.

Getting stuck is something that most learners experience at one time or another. In Paper III we investigate how successful students handle these situations. The results show that the students use a large number of different strategies to get unstuck and continue their learning. Many of the strategies involve social interaction with peers and others.

In Papers IV, V, and VI we study what students experience as being key and threshold concepts in Computer Science. The results show that understanding particular concepts indeed affect the students greatly, changing the way they look at Computer Science, their peers, and themselves.

The two last papers, Papers VII and VIII, investigate how researchers, teachers and students view concurrency. Most researchers/teachers claim that students have difficulties because of non-determinism, not understanding synchronization, etc. According to our results the students themselves do not seem to think that concurrency is significantly more difficult than any other subject. Actually most of them find concurrency to be both easy to understand and fun.

Sammanfattning

Programmering har en central roll i datavetenskapliga utbildningar. Många anser att programmering är svårt att lära sig. Ett stort antal studier har undersökt vad som orsakar dessa svårigheter och hur det är möjligt att övervinna dem. Denna avhandling är en del av denna forskning. Artiklarna i avhandlingen undersöker vilka problem som studenterna stöter på under sina programmeringsstudier.

Artikel 1 beskriver hur studenter använder sig av annoteringar vid problemlösning. Resultaten visar att studenter som gör många annoteringar tenderar att prestera bättre. Resultaten antyder också att det kan finnas kulturella skillnader i hur ofta annoteringar används.

Studenter har inte bara problem vid programmering, de har också problem med att utforma programvarusystem. Även sistaårsstudenter misslyckas till stor del att utforma lösningar för relativt enkla system. Resultaten i Artikel II visar att majoriteten av studenterna inte kommer längre än en omformulering av problemet.

Att inte förstå ett koncept eller en specifik detalj är något som alla studenter stöter på då och då. I Artikel III undersöker vi hur framgångsrika studenter hanterar en sådan situation. Resultaten visar att studenterna använder sig av ett stort antal olika strategier för att få en förståelse för konceptet/detaljen. Många av de redovisade strategierna bygger på en social interaktion med andra.

Artiklarna IV, V och VI utforskar vad studenterna uppfattar som nyckelkoncept inom datavetenskap och hur förståelsen av dessa koncept påverkar dem. Resultaten visar att förståelsen av vissa specifika koncept kan göra att studenterna ändrar hur de ser på datavetenskap, kollegor och sig själva.

I artiklarna VII och VIII undersöker vi hur forskare, lärare och studenter ser på de problem studenter har vid jämlöpande programmering. De flesta forskare och lärare hävdar att studenterna har problem med att förstå icke-determinism, synkronisering, etc. Våra resultat visar dock att studenterna inte själva tycks anse att jämlöpande programmering är signifikant svårare än andra ämnen. Tvärtom, de flesta anser att jämlöpande programmering är både lätt att förstå och roligt.

Preface

This thesis consists of an introduction to the research area and the following papers:

Paper I **Questions, Annotations, and Institutions: observations from a study of novice programmers** in *Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*.
Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä. p11–19, 2004

Paper II **Can graduating students design software systems?** in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*.
Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Carol Zander. p403–407, 2006

Paper III **Successful students' strategies for getting unstuck** in *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*.
Robert McCartney, Anna Eckerdal, Jan Erik Moström, Kate Sanders, Carol Zander. p156–160, 2007

Paper IV **Concrete examples of abstraction as manifested in students' transformative experiences** in *ICER '08: Proceeding of the Fourth international Workshop on Computing Education Research*.
Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, Carol Zander. p125–136, 2008

Paper V **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*.
Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Lynda Thomas, Carol Zander. p332–336, 2008

Paper VI **Student transformations: are they computer scientists yet?** in *ICER'09: Proceedings of the fifth international workshop on Computing education research workshop*
Carol Zander, Jonas Boustedt, Robert McCartney, Jan Erik Moström, Kate Sanders, Lynda Thomas. p129–140, 2009

Paper VII **Learning concurrency — what's the problem?** Jan Erik Moström 2011

Paper VIII **Students' experience of learning concurrency** Jan Erik Moström 2011

Contribution

A list of the author's contribution to the articles previously listed is described below.

It should be noted that most of the authors in articles 1–6 have been involved in a 7-year long research collaboration – the group usually goes under the name "The Sweden Group" – which has been quite successful. One of the habits of this group is to have long and regular discussions about the data being analyzed. During these discussions, ideas are raised, modified, and re-modified; data and results are examined and re-examined; and text is written, edited and later re-edited. In fact the collaboration has been so "tight" that the group members have had difficulties to point out what each individual has contributed – even directly after a paper has been submitted for publication. The "joint collaboration" in this group has truly been "joint" and thus much of my contribution has been to participate in the discussions and put forward my ideas and opinions.

Paper I The data used in this paper was originally collected for a Workshop at ITiCSE 2004. The first observations that students annotated their solutions in various ways was done by me, the analysis in this paper was done jointly by the authors.

Paper II The data analyzed in this paper was collected by approximately 20 people as a part of a larger international project. It was my comments on the data that steered the analysis in its final direction. Otherwise, the analysis and writing was done jointly by the five authors.

Paper III Data collection was done by all authors and one additional person. Analysis and writing was done together jointly by the five authors.

Paper IV Data collection was done by all authors. As usual the data and analysis was discussed within the group; my contribution was mainly in the sections on 'modularity' and 'data abstraction'.

Paper V Data collection was mainly done by two of the authors - Carol Zander and Lynda Thomas. Analysis of data and writing was done jointly by all authors.

Paper VI This paper used the same raw data as paper 5. Much of the work was done jointly by the authors; my main contribution was in the section "Identity" together with Carol Zander.

Paper VII All work done for this paper was done by the author.

Paper VIII All work done for this paper was done by the author.

Other papers by the author

- **A multi-national, multi-institutional study of student-generated software designs** in *Koli Calling 2004 - Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*.
  Sally Fincher, Marian Petre, Josh Tenenberg, Ken Blaha, Dennis Bouvier, Tzu-Yi Chen, Donald Chinn, Stephen Cooper, Anna Eckerdal, Hubert Johnson, Robert McCartney, Alvaro Monge, Jan Erik Moström, Kris Power, Mark Ratcliffe, Anthony Robins, Dean Sanders, Leslie Schwartzman, Beth Simon, Carol Stoker, Allison Elliot Tew, and Tammy VanDeGrift. p20–27, 2004.

- **Cause for alarm?: A multi-national, multi-institutional study of student-generated software designs**. *Technical Report 16–04, Computing Laboratory, University of Kent, Canterbury, UK*.
  Sally Fincher, Marian Petre, Josh Tenenberg, Ken Blaha, Dennis Bouvier, Tzu-Yi Chen, Donald Chinn, Stephen Cooper, Anna Eckerdal, Hubert Johnson, Robert McCartney, Alvaro Monge, Jan Erik Moström, Kris Power, Mark Ratcliffe, Anthony Robins, Dean Sanders, Leslie Schwartzman, Beth Simon, Carol Stoker, Allison Elliot Tew, and Tammy VanDeGrift. 2004.

- **A multi-national study of reading and tracing skills in novice programmers** in *SIGCSE Bulletin*.
  Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. p119–150, v36, 2004.

- **Comparing Student Software Designs Using Semantic Categorization** in *Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education*.
  Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. p57–64, 2005.

- **Take Note: the Effectiveness of Novice Programmers' Annotations on Examinations** in *Informatics in Education*.
  Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä p69–86, v4n1, 2005

- **Students designing software: a multi-national, multi-institutional study** in *Informatics in Education*.
  Josh Tenenberg, Sally Fincher, Ken Blaha, Dennis Bouvier, Tzu-Yi Chen, Donald Chinn, Stephen Cooper, Anna Eckerdal, Hubert Johnson, Robert McCartney, Alvaro Monge, Jan Erik Moström, Marian Petre, Kris Powers, Mark Ratcliffe, Anthony Robins, Dean Sanders, Leslie Shwartzman, Beth Simon, Carol Stoker, Allison Elliot Tew, and Tammy VanDeGrift. p143–162, v4n1, 2005.

- **Putting threshold concepts into context in computer science education** in *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*.

Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. p103–107, 2006.

- **Categorizing student software designs, methods, results, and implications** in *Computer Science Education*.
  Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander p197-209, v16n3, 2006.

- **What is the Word for "Engineering" in Swedish: Swedish Students Conceptions of their Discipline** in *Technical Report 2007-018, Department of Information Technology, Uppsala University*.
  Robin Adams, Sally Fincher, Arnold Pears, Jürgen Börstler, Jonas Boustedt, Peter Dalenius, Gunilla Eken, Tim Heyer, Andreas Jacobsson, Vanja Lindberg, Bengt Molin, Jan Erik Moström, and Mattias Wiggberg. 2007.

- **Threshold concepts in computer science: do they exist and are they useful?** in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*.
  Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. p504–508, 2007.

- **From Limen to Lumen: computing students in liminal spaces** in *ICER '07: Proceedings of the third international workshop on Computing education research.*
  Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. p123-132, 2007

- **Debugging assistance for novices: a video repository** in *SIGCSE Bulletin*.
  Beth Simon, Sue Fitzgerald, Renée McCauley, Susan Haller, John Hamer, Brian Hanks, Michael T. Helmick, Jan Erik Moström, Judy Sheard, and Lynda Thomas. p137–151, v39n4, 2007.

- **What's the problem? Teachers' experience of student learning successes and failures** in *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*.
  Arnold Pears, Anders Berglund, Anna Eckerdal, Philip East, Päivi Kinnunen, Lauri Malmi, Robert McCartney, Jan-Erik Moström, Laurie Murphy, Mark Bartley Ratcliffe, Carsten Schulte, Beth Simon, Ioanna Stamouli, and Lynda Thomas. p207–211, v88, 2007.

- **Evaluating OO example programs for CS1** in *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education.*
  Jürgen Börstler, Henrik B. Christensen, Jens Bennedsen, Marie Nordström, Lena Kallin Westin, Jan Erik Moström, and Michael E. Caspersen. p47–52, 2008

- **Transitioning to OOP - A Never Ending Story** in *Reflections on the teaching of programming*. Edited by Michael Kölling, Jens Bennedsen and Michael E. Caspersen.

Jürgen Börstler, Marie Nordström, Lena Kallin Westin, Jan Erik Moström, and Johan Eliasson. p80-97, 2008.

- **An Evaluation Instrument for Object-Oriented Example Programs for Novices** in *Technical Report UMINF 08.09, Umeå University*.
  Jürgen Börstler, Marie Nordström, Lena Kallin Westin, Jan Erik Moström, Henrik B. Christensen, and Jens Bennedsen. 2008

- **DCER: sharing empirical computer science education data** in *ICER '08: Proceeding of the Fourth international Workshop on Computing Education Research*.
  Kate Sanders, Brad Richards, Jan Erik Moström, Vicki Almstrum, Stephen Edwards, Sally Fincher, Kat Gunion, Mark Hall, Brian Hanks, Stephen Lonergan, Robert McCartney, Briana Morrison, Jaime Spacco, and Lynda Thomas. Lynda p137–148, 2008.

- **Threshold Concepts in Computer Science: A Multi-National Empirical Investigation** in *Threshold Concepts within the disciplines*.
  Carol Zander, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Kate Sanders. 2008.

- **Learning Computer Science: Perceptions, Actions and Roles** in *European Journal of Engineering Education*.
  Anders Berglund, Anna Eckerdal, Arnold Pears, Philip East, Päivi Kinnunen, Lauri Malmi, Robert McCartney, Jan Erik Moström, Laurie Murphy, Mark Ratcliffe, Carsten Schulte, Beth Simon, Ioanna Stamouli, and Lynda Thomas. p327–338, v34n4, 2009

- **Liminal Spaces and Learning Computing** in *European Journal of Engineering Education*.
  Robert McCartney, Jonas Boustedt, Anna Eckerdal, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. p383-391, v34n4, 2009.

- **Computer science student transformations: changes and causes** in *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*.
  Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. p181–185, 2009

- **Threshold Concepts in Computer Science: An Ongoing Empirical Investigation** in *Educational Futures: Rethinking Theory and Practice*.
  Lynda Thomas, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, and Carol Zander. p241–258, v42, 2010.

Acknowledgements

It comes without saying that I have a lot of people to thank for a lot of different things.

The first person I would like to thank is Jürgen Börstler. I remember sitting in my office, trying to figure out how my newly registered company would make some money, when Jürgen walked in, handed me a paper and said "Here is a course/project about Computer Science Education Research that NSF arranges in Seattle. I think you should apply". I did. And that was the start of me switching interests and getting to know a lot of interesting people. I later became an official Ph.D. student (again) and Jürgen became my advisor a few years later. And here I am, all because of Jürgen – and if anyone is interested: my company did not make a lot of money.

Secondly, I would really like to extend my deepest thanks to the group of people I have been working with for several years: Jonas Boustedt, Anna Eckerdal, Robert McCartney, Mark Ratcliffe, Kate Sanders, Lynda Thomas, and Carol Zander. Thanks for putting up with me, and for teaching me so much.

An extra "Thank you" goes to Kate and Carol. They have read my thesis, corrected my Swinglish, fixed numerous spelling mistakes, and too many grammatical errors. However, when you read the thesis you will undoubtly see some errors — I can proudly say that they are all my own! As tradition has it, several last minute changes/additions were made to the text after Carol and Kate had finished their work.

I would also like to thank the people in the Didaktitum research group at my department: Marie, Lena, Pedher and Johan. My thanks also goes to the people involved in the NSF founded projects Bootstrapping and Scaffolding — with a special thanks to Sally, Marian and Josh.

And to the Computing Science Department I would like to say: thanks for making it possible for me to do this work. I would like to especially thank Mattias, who did a lot of the work that I should have done these last few months.

There are also two groups of people that probably have no idea that I have been working on this thesis. Nonetheless, you have kept me reasonable sane and have provided me with many hours of pain, sweat and laughs. In the first group I would especially like to thank Yngve that tricked me into the whole thing, but also Anders, Andreas, Andreas, Anna, Anna, Annika, Christer, Daniel, Emil, Emma, Erik, Fredrik, Fredrik, Gustav, Hasse, Ida, Jens, Joakim, Johan, John, Johnny, Jonas, Kenneth, Malin, Maria, Ola, PA, Pelle, Robin, Sanna, Tobias, Wille and all the rest of you at UBK.

In the second group I would like to thank: Anders, Bernt, Bertil, Bosse, David, Fredrik, Jonny, Jonas, Jonas-Erik, Jonathan, Kenneth, Kent, Lage, Lars, Lars, Leif, Micke, Roger, Rune, Sebastian, Stig, Sune, Tomas, Tony och Örjan. I would also like to thank Ragnar, Roland and Rolf although they are not among us any more.

And of course a very special “Thank You” to Annika, Hannes, Erik, Martina and the rest of my family.
— jem, now signing out

Contents

Description of papers

Paper I - Questions, Annotations, and Institutions: observations from a study of novice programmers

This paper explores the connection between the way in which students annotate problems and how well they perform. The results show that successful students tend to annotate their work more than less successful students, a result that is confirmed by other studies. However, students at two universities do not follow this pattern and instead annotated their work sparingly. This becomes even more interesting when we observe that these two universities are located in two neighboring countries and have a common Computer Science history.

The results also show that the annotations vary depending on the type of question. For example: fixed-code questions got more annotations classified as tracing, while elimination type annotations were favored in skeleton-code questions.

While these results seem to be "common-sense results" they open up new interesting questions:

- If annotations improve student results, should we not teach how to use annotations? The paper cites a study where students are encouraged to use annotations, but they still fail to take full advantage of them. Is this a typical behavior or can we encourage students in some other way?

- Some of the annotations are used to track the dynamic behavior of a program. This is interesting since most of the programming languages we use today are static textual languages that usually can not show what happens dynamically when the program is executed. Do these student notations indicate that we need a notation for the dynamic nature of a program? Would such a notation make a difference for the students? (note that "notation" does not refer to program visualization tools, run-time debuggers, etc)

- The results show a connection between annotations and student performance, except for students at the two universities mentioned above. These students perform at an equal level to the students at other universities but use annotation more sparingly. This might indicate that there are cultural differences in how Computer Science is taught/learned around the world.

Can we take advantage of the differences in some way (if they actually exist)?

Unfortunately the data that were collected for the study did not include information that allowed us to pursue these questions in more detail.

Paper II - Can graduating students design software systems?

Some studies [35, 29] report on novice difficulties in completing simple programming problems. Lister et al. [29] even suggest that students have problems reading/understanding basic code snippets. This paper makes a similar investigation into graduating students' ability to design software.

The participants in the study were asked to design a "super alarm clock" to help students manage their sleep patterns. The resulting designs were categorized into six categories. This categorization shows that only 38% of the students produced something that at least could be considered to be a first step towards a finished design. More than 60% failed to make any significant progress towards a design. Some of these, quite depressing, results might be explained by the students' inexperience with under-specified tasks similar to the one used in the study. However, the results indicate that even graduating students have difficulties in designing software.

It is also worth noting that academically high-performing students did not produce the best designs. Instead many of them produced designs that were categorized as "First step" or below. The best designs were in fact produced by students with an average grade.

Based on these results it seems fair to say that we as educators have failed to teach students how to design software. While some of the failures might be explained by the students lack of experience with these types of problems it does not explain everything. It might be prudent to carefully consider what goals we set for our students and whether we manage to help the students reach them.

Paper III - Successful students' strategies for getting unstuck

We all get stuck at times when learning something new. It can be that we do not understand a specific concept, a procedure for doing some task, or we might lack some knowledge that is necessary to learn a new concept. The end result is the same: we get stuck in our learning and we have to find a way to get unstuck, for example, by finding a work-around to avoid the problem or by asking friends and colleagues for help.

This paper investigates how students get unstuck. The results show that students use a large number of strategies. None of the strategies found should surprise the experienced teacher, in fact many of the strategies are similar to the advice commonly given for how to study successfully. However, it is interesting to note that social interaction plays an important role for many students.

This brings up the very interesting question of how unsuccessful students handle situations where they get stuck. Do these students fail their studies

because they can not get unstuck? If so, what kind of strategies do the unsuccessful students use? Further, if students fail because they get stuck and fail to get unstuck, what can we do to help them acquire learning strategies that will help them to get unstuck?

It is worth noting that social interaction is important for the students. Is this something we can encourage by increasing the time students work together. An important question is also how this affect students participating in distance education courses, is there a difference in how they interact with their peers. And if that is the case, how does this affect their academic performance.

The list of different strategies also serves as a nice reminder for teachers of the importance of showing different methods for attacking a problem.

Paper IV - Concrete examples of abstraction as manifested in students' transformative experiences

Many Computer Science educators consider abstraction to be a key concept in Computer Science. This paper shows that this is indeed true: abstraction plays an important role for many students when learning to program. However, students do not talk about abstraction per se, in fact they might not mention abstraction at all. Instead they describe transformative events that occurred due to abstraction.

It is also worth noting that students might have reached this transformative event either by first gaining an abstract understanding of the subject, followed by practical experience, or vice versa. Once again, we see the importance of both allowing students to view a subject from a number of different angles, and of gaining practical experience.

Paper V - Student understanding of object-oriented programming as expressed in concept maps

This paper reports on an investigation of student understanding of object-oriented concepts. Instead of more traditional data collection methods like interviews, questionnaires, etc., this study used concept maps. This allowed the participants, given some "seed concepts," to freely come up with concepts and associate them with the existing ones.

The results contain some interesting information: students seem to associate class and behavior more strongly than class and data, message passing between objects is not mentioned at all, abstraction, polymorphism and encapsulation were rarely mentioned, and it is unclear whether students really understand the connection between class and objects.

The results show that some concepts that many consider to be a fundamental part of object orientation are not what the students immediately think of. A possible method for improving the results could be to let these concepts have a more prominent role in our courses, making the students more aware of them.

Paper VI - Student transformations: are they computer scientists yet?

The Sweden Group has investigated Threshold Concepts in relation to Computer Science in several studies. One of the properties of a Threshold Concept is that once it is understood it permanently changes the learner's view of the subject. (See Section 4.2 for more details about Threshold Concepts.) In this paper we investigate what transformative events students have experienced and how these events affected them personally.

Students report on a large number of different concepts that changed their view of Computer Science and different types of transformation. Some students changed their habits of thinking about and solving problems, others changed the way they looked at Computer Science in a larger context, while still others indicated that they had changed the way they looked at themselves.

One of the lessons learned from this study is that learning to program is not only a matter of learning facts, syntax and process. It is also the process of going from being a novice to a recognized member of a professional community.

Paper VII - Learning concurrency — what's the problem?

It is common wisdom among Computer Science teachers that concurrency is a hard subject to learn and that students have difficulties in learning it. Interestingly enough there are other teachers/researchers who claim that students have no difficulties in learning concurrency and that it should be a part of early programming courses (or even to be the foundation for all programming courses).

In this paper we survey what has been written about students and their learning of concurrency. The results indeed show that there are different opinions on how difficult learning concurrency is. (See Section 5.4 for more information.) But more importantly the paper also shows that only limited empirical evidence exists for various claims.

The results also include a categorization of what is said to cause students their problems. Unfortunately, these categories include a large number of the concepts and techniques used when designing and implementing concurrent systems.

Perhaps the most important observation in this paper is the lack of empirical data on what is causing problems for the students. Nevertheless, as some of the studies show there is much to be learned by doing basic research.

Paper VIII - Students' experience of learning concurrency

The previous paper investigates what teachers and researchers have to say about students and their problems in learning concurrency. In this paper we report on a interview study with students who have taken at least one course where concurrency is taught.

The results, described in more detail in Section 5.5, show that the students had a positive view of concurrency. They did not find it to be significantly more

difficult than any other CS subject. Also, to some the added complexity only acted as additional challenge making it even more interesting.

What came as a surprise was that there were few spontaneous mentions of the need of of code/process visualization, something that was quite popular in the articles surveyed in the paper described above.

Chapter 1

Introduction

When students learn to program it is not uncommon to hear comments on how difficult this is. The author's personal experience include comments like "This is impossible", "Programming is sooo difficult", "I can't understand this", "I will never learn to how to program", "This is so frustrating" and the very generic "Arrrrgh" (sometimes followed by the sound of a fist hitting a keyboard, or a table). This could, of course, be the result of failing as a teacher, but judging from the numerous articles describing student's problems in learning to program, many teachers seem to have the same experience.

Since "programming" includes a large number of different activities and various concepts, the studies in the literature describe a large number of different problems of student's learning to program. Most of these studies have focused on novices and the problems they experience; we can find articles where authors discuss students understanding of basic language constructs [29], how the programming paradigm affects students understanding [46], how students debug programs [51], and so on.

One of the concepts seen to cause problems is concurrency. It is suggested by many authors that the introduction of multiple execution threads is something that students find difficult. There are many suggestions as to what causes these difficulties [37] and what programming language, tool, and/or course design to use to make it easier for students to learn.

Interestingly, another group of authors writes that concurrency is not something that students find difficult and that it is something that can be introduced early in the curricula. It can be somewhat difficult to determine if one of these groups is "right" since they both argue convincingly for their case.

This thesis can be seen as a summary of my attempts to better understand the problems students experience when learning to program.

1.1 Outline of the thesis

This thesis is outlined as follows: In Section 2, I give a short general description of the activities that "programming" includes. Section 3 provide an overview of Computer Science Education Research, followed by Section 4 that describes the research I have been involved with. The concept of concurrency and students' problem with the subject is explored in Section 5. I make some general conclusions in Section 6. This is followed by the papers previously described.

Chapter 2

Programming

How can we describe what commonly is referred to as "programming"? While to many non-programmers it might seem to be a single activity, in reality it can be a very diverse series of activities aimed at solving a wide variety of problems. There is, for example, a huge difference between designing and implementing a system for weather forecasts and a small script for cleaning up log files. Assume that we need to come up with one sentence describing "programming"; our attempt to do this resulted in:

> Programming is the act of understanding a problem, formulating a solution, and writing down the solution in such a way that a computer can use the solution to solve the problem.

There are other issues to consider, for example, maintainability, but let us ignore those for the moment. Depending on the problem, these activities might involve a single programmer or several hundred individuals with various specialities. For the sake of brevity, assume that a single individual is responsible for the complete solution. This means that this individual, the programmer, first needs to understand the problem domain; for example, when designing/implementing a system for accounting, the programmer must understand accounting and how accountants work. Once the problem is understood, the programmer will use standard problem-solving techniques to break the problem down into small enough subproblems that can be solved using the tools and knowledge available to the programmer. In the final step, the programmer needs to communicate the solution in such a way that a computer can follow the instructions. The most common method today is to write down a very detailed set of instructions using a textual language such as C, C++, Java, Python, or PHP. The reality, of course, is much more complex than this very simplified description, but it catches the three main activities in programming.

The human ability to solve problems is something that has been of interest to researchers in Cognitive Science and we will not go into the details of problem solving here; we recommend that the interested reader read an introductory book on Cognitive Science [14] to get an overview. One problem

solving strategy is to divide a problem into smaller sub-problems that can be solved individually. These solutions can then be combined to solve the complete problem. This "divide-and-conquer" strategy can be found in many Computer Science textbooks. In fact, most programming languages are designed in a way that encourage the programmer to use "divide-and-conquer" – for example by using procedures, functions, classes, modules, etc.

One important part of programming is writing down the envisioned solution in a form that can be translated into something that the computer can execute. Although this might sound like a simple task, it has been the source of much discussion, for example, on conflicts between programming and natural languages [3], on comparing different paradigms [46], whether novices understand even simple code snippets [29], or programming in general [17]. Not only have different language paradigms like object-oriented and logic programming been suggested and discussed, but also what the notation of the different paradigms should look like. Traditionally most programming languages have used a textual notation but there have been suggestions to use graphical programming languages like Agentsheets[1] [47], StageCast [9] or Scratch[2] [32] or even animated programming like ToonTalk [20] to make programming more accessible.

Some research has focused on the learning situation, how students learn, how they become professionals, etc. Some of this research has resulted in insights on how well students have learned to program [35, 29, 30], the importance of good examples [44], students' understanding of correctness [25, 21], cultural conflicts between students and professionals [26], etc.

Despite all the effort that researchers and teachers have put into understanding and helping students, there is still much work to be done.

[1] `http://www.agentsheets.com/`
[2] `http://scratch.mit.edu/`

Chapter 3

Computer Science Education Research

Computer Science Education (CSEd) research is somewhat different than many other research areas in Computer Science. In many areas it is possible to base research on collecting and analyzing quantitative data, usually using statistical methods. While quantitative data certainly can be collected and analyzed doing CSEd research, it has a number of limitations. Perhaps the biggest problem is that a statistical analysis might indicate that there are a number of problems and tell us how common they are. But with education research, we are usually not interested in the manifestations of student problems; instead we are interested in *why* they experience these difficulties and what is causing them – we want to understand how the students think and why they make the mistakes they do, not just that they do make mistakes.

Similarly, the fact that that 24% of the students have problem A, 16% problem B, 21% problem C, etc., does not give us enough information. We can not discard problem E just because "only" 11% of the students have this problem. Instead we want to know *what kind of problems* students encounter. Once we know this, we can adapt our presentation of the subject to take these issues into consideration, hopefully making it easier for the students to learn. In general, it is less interesting to know how many students experience certain problems, compared to knowing what kind of problems they experience and why.

It goes without saying that it is very difficult to objectively measure how someone thinks; literature in Cognitive Science shows the many difficulties researchers have in trying to understand how humans think. Instead we must turn to qualitative data collection. To someone trained in quantitative research these methods might seem "strange", "unscientific", and "subjective" since they do not necessarily involve results that are quantifiable or easy to generalize. Qualitative researchers aim to gather an in-depth understanding of human behavior and the reasons behind such behavior. Qualitative methods are designed to get at the why and how, as opposed to the what, where, and when. Researchers

in psychology, social sciences, etc., have extensive experience in conducting this type of research, experience we can take advantage of.

3.1 Collecting data for CSEd research

It is our experience that often both quantitative and qualitative data are collected during a CSEd study. The quantitative data often contains basic background information such as student age, gender, year of study, type of educational institution, etc. However, the vast majority of data comes in the form of written essays, collected drawings/diagrams and, most importantly, recordings of interviews. Below we will discuss these three types of data in greater detail.

Essays

One very attractive feature of essays, at least to the researcher, is that they are easily converted to a form that can be analyzed. In fact in many instances they come in electronic form and can directly be used for analysis. Unfortunately, this data collection method also has some drawbacks, for example:

- It can be difficult or impossible to follow up on issues raised in the essays. This can be extremely frustrating to the researcher, not being able to follow up on issues mentioned or ask for clarifications.
- The student who writes the essay can limit and/or censor him/herself because of the time/effort required to write a detailed description. Writing down a detailed explanation can take quite some time and it is our experience that written comments on questionaries and essays tend to be quite terse.

However, while these issues limit the usefulness of the data, the method also has advantages. Using essays it also becomes possible to collect data from students that would not have been available for interviews.

Sketches

It is not uncommon for a student to make spontaneous sketches during an interview or during some kind of problem solving activity [29]. Sketches can be difficult to analyze since they are usually of a dynamic nature. For example, sketches made during an interview tend to evolve over time. This makes it difficult to analyze how the the sketch looked like at a certain point in the interview. Similarly, sketches collected at the end of problem solving exercise are different to analyze since it is difficult to know how they looked like at a certain point of time in the problem solving activity. In other words, the dynamic nature of sketches makes them difficult to analyze.

However, by using something like an Anoto pen[1] or Explanogram technology[45], it becomes possible to study the dynamics of a sketch. We have had the op-

[1] `http://www.anoto.com/` visited 2011-06-13

portunity to try to analyze that same sketches with and without the use of Explanograms and we found it fascinating how differently we interpreted the static sketches compared to the dynamic version that Explanograms offered us[2]. We recommend that anyone who plans to collect sketches as a part of their study use these or similar technologies.

Interviews

Interviews are perhaps the most important tool we have to collect qualitative data on student experience of learning. A commonly used type of interview is the semi-structured interview where the interviewer has a predefined set of topics to cover. While the interviewer has a script to follow there is no need to follow it religiously, instead the interview should be flexible and allow the interviewee to control the interview to a certain extent. The interviewer should also take the opportunity to ask follow-up questions in an attempt to get more details. The interviewer should also make sure that all questions have been covered before ending the interview.

A good interviewer understands his/her role during the interview, recognizing how different types of questions affects the interviewee, how the intonation, choice of words, body language, etc., all affect the outcome of the interview. During analysis it is important to consider which role the interviewer played in the interview, for example, if the interviewer introduced any bias or in some other way influenced the interviewee.

3.2 Analysis methods

As with any experimental work, it is important to plan the experimental setup carefully so that it fits with the approach that has been selected for doing the data collection and analysis. Popular analysis methods includes

- Grounded Theory
- Narrative Research
- Phenomenology
- Content (or Thematic) Analysis
- Ethnography
- Case Study
- Phenomenography

[2]This was only done as an informal experiment and did not result in any comparison between the methods.

See for example [7, 4] for more information. For the novice qualitative researcher it can sometimes be difficult to differentiate these methods and understand when to use which method. In our research we have not used a "pure" method, instead we have analyzed the data in a way that is inspired by Grounded Theory and Phenomenography. Our workflow can shortly be described as

1. Collect raw data - the majority being semi-structured interviews.
2. Transcribe.
3. Read the interviews and look for various themes/categories.
4. Re-read the interviews and refine the themes/categories. Repeat this step until are no further changes in the themes/categories.
5. Analyze the themes/categories with regard to the research question(s).

However, it is important for the researchers to be aware of the limitations that come with the choice of data collection methods and analysis. As Fincher el al. [15] point out, there is interesting information to find if we look outside our normal ways of collecting data.

Chapter 4

Student problems in learning to program

Computer Science Education research has to a large extent focused on novice programmers and the problems they experience when first learning to program. Some of the work focus on syntactical/semantical issues [10] and the use of programming plans/goals [54, 53]. There has also been extensive work done to investigate alternative ways of programming, for example programming by demonstration [8], programming by example [20, 42], programming with examples [41], graphical programming languages [9, 47, 32], and animated programming languages [19]. How programming can be used by the computer users in general [43], for example in spreadsheet applications where users are able to define how calculations should be done. We can also find a large number of studies on specific topics related to learning to program. Some examples include studies on examples for object-oriented programming [44], how students handle new concepts [28], how previous knowledge affects learning [56, 52], various factors that affect how students learn to program [59], how students read and understand code [29, 31, 16, 58], and the transition from being a student to a professional [1].

The author of the present thesis has mainly studied the following areas:

- Understanding how to program
- Threshold Concepts
- Social aspects
- Concurrency

This section describes the first three subjects while concurrency is explored in more detail in Chapter 5

4.1 Understanding how to program

As we have mentioned earlier, there has been much work on helping students learn how to program. However, student problems persist and even tasks that most CS teachers would consider to be fundamental, such as reading code, can pose considerable problems for the students [29]. Lister et al. summarizes their results:

> The McCracken group established that many first-year programming students cannot program at the conclusion of their introductory courses. While a popular explanation for that inability is that students cannot problem-solve, in the strong five-step sense defined by the McCracken group, this working group has established that many students lack knowledge and skills that are a precursor to problem-solving. These missing elements relate more to the ability of students to read code than to write it. Many of the students manifested a fragile ability to systematically analyze a short piece of code.

These results indicate that students' difficulties are, partially, caused by the inability to read and understand code. It is also worth noting that students who annotated their work did better than those who did not. McCartney et al. [34] show that while all students seem to benefit from annotating their work, different types of annotations are used for different types of problems. Our study used two different types of multiple choice questions:

Fixed-code. The student is given a code fragment and answers questions about what is the result of the execution of the code.

Skeleton-code. Here the students are presented with an incomplete code snippet and are asked to complete the code by selecting the correct code snippet.

For the fixed-code questions, students tended to use some kind of tracing to find the correct answer, while the process of elimination was a more popular strategy for skeleton-code questions. The study does not find evidence to explain these differences but the authors suggest three possible reasons: 1) too much work is required to do tracing for skeleton-code problems, 2) fixed-code questions are less abstract and thus more suitable for tracing, and 3) there is a lack of a notation to represent the more complex code snippets used for skeleton-code questions.

Not only do students perform on a less than expected level in reading/writing simple programs, they also have problems in designing programs. In Eckerdal et al. [12] we describe a study where 149 graduating student design solutions were categorized. The results indicate that a large number of the students, 62% of the observations, of the graduating students produce designs that the authors categorize to be less than a "First Step". Only 2% of the observations were considered to be a "Complete" design. The study also indicates that students'

grades have little relationship to their produced design as the designs of many of top performing students were classified to be below a "Partial" design.

4.2 Threshold Concepts

The idea of Threshold Concepts [36] has received interest from the CSEd community [5, 11, 13, 33, 48, 49, 57, 60] with the hope that it would make it possible to define a number of *core* concepts that students must master in the discipline. This would enable educators to concentrate their efforts on these concepts, resulting in better understanding from the students. Meyer and Land [36] defines Threshold Concepts as being:

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: students experience these concepts as conceptually difficult, alien, and/or counter-intuitive.

Often boundary markers: they indicate the limits of a conceptual area or the discipline itself.

The motivation behind Threshold Concepts is the "aha moment" that most of us have had, i.e., we come to understand a concept that has troubled us for some time and once we have gained this understanding, other things become clear. When asked about their opinion of Threshold Concepts many educators found them very interesting [5]. In the Sweden Group, we have investigated Threshold Concepts in Computer Science thoroughly [11, 5, 13, 60, 33, 40] We identified two possible Threshold Concepts:

> 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 [5, p. 508].

In a later paper [39], the abstraction of specific concepts – for example, modularity, data abstraction, inheritance, polymorphism, design patterns, complexity – was added. However, it is difficult to find evidence of these as Threshold Concepts and they do not seem to be universal in the sense that every student experiences them.

Moreover, it is problematic that the discovered concepts have such broad scope. For example, the notion of object-oriented programming as a Threshold

Concept is almost useless since object-oriented programming is such an encompassing topic – it is usually the subject for several courses – that it becomes impossible to know where to concentrate the teaching effort. In Sanders et al. [50], concept maps were used to investigate students' understanding of the concepts that make up object-oriented programming. Students were found to confuse "class" and "instance" but students generally connected classes with both data and behavior to various degree. They have a static model of object-orientation showing no interaction among objects in a program.

4.3 Social aspects

Some authors point out that it is not only facts and skills that students have to learn before they can claim to be Computer Scientists, they also have to become a part of the Computer Science Community by learning the language, standards, and way of thinking that is the accepted behavior in the community.

Ben-David Kolikant [23] describes the culture clash that happens when old-timers in the informal programming culture – the students – meet the academic programming culture – the teachers. She suggests that these cultural meetings should be considered fertile zones of cultural encounter where bridges are built between the students' current culture and the target academic culture. Kolikant suggests that students' and teachers' different standards of correctness [25, 21] might be explained by this cultural clash [26]. The following quote characterizes this clash:

> In contrast, we found that CS students assert and exercise rights to grant legitimacy to activities proposed by the teachers and will not make the effort needed to learn advanced subjects until such legitimacy has been granted. We believe that the standing of teachers in CS is different from teachers' standing in other subjects: Students challenge the demands set by the teachers because they know how to productively solve problems on the computer, although they have different ideas on what counts as a satisfactory solution.

Zander et al. [61] describe a similar progression from computer user to computer science student to computer science professional is described. However, it is difficult to say to what extent this affects the students' learning.

Chapter 5

Concurrency

It is safe to say that programming is not something that comes naturally to everyone. Thus it is not surprising to see that many studies have been focused on novices and the problems they experience, while less attention has been given to the problems experienced by more advanced students – despite that many new and complex concepts are introduced in the later years. Concurrency is one of the concepts that traditionally is introduced later in the curricula and is often considered difficult.

5.1 What is concurrency?

The main difference between a serial and a concurrent system is that the serial system has a single thread of execution whereas a concurrent system has multiple threads, see for example [6, 18, 55]. Figure 1 shows a traditional serial program. The dotted rectangle symbolizes the program code, while the arrowed line shows

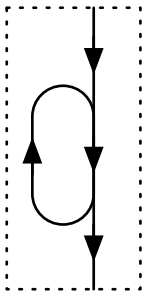

Figure 1: A serial program, the arrowed line shows the execution path from beginning to end. The loop in the middle symbolizes a loop in the code.

the execution path – sometimes called a 'thread'. We can see how a traditional program runs from start to end along a single path. This is in contrast to a

concurrent program that allows the execution of several different paths at the same time, see Figure 2.

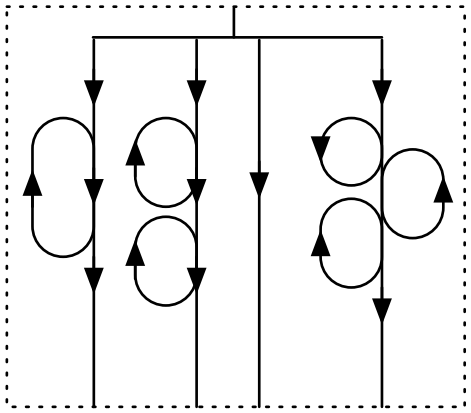

Figure 2: A concurrent program that splits a single execution path into several concurrent paths.

A second noticeable thing is that the different threads in a concurrent system work together towards a common goal. To be able to achieve this goal it is necessary to coordinate the activities of the threads; in other words, there is a need for a mechanism that allows the threads to communicate with each other. Several methods for how to communicate have been suggested and we will take a closer look at one of them later.

5.2 What problems do concurrency solve?

In early computer systems, concurrency was simulated by letting one processing unit quickly switch between different threads. It turns out that despite the extra strain these switches put on the processing unit, the total performance of a system can increase. How can this be? Input/Output (I/O) operations are usually very slow compared to other execution. This means that a program that has many I/O operations spends a considerable amount of time waiting for these operations to complete. By switching to another thread and letting it execute during these periods, we can achieve a total increase in performance[1], Figure 3 illustrates this phenomen.

If we have several processing units available, we can also achieve higher performance by dividing a problem into several sub-problems and letting the different units solve each part. The sub-solutions are then combined to provide the complete solution. For example, this is a common technique when making large matrix calculations as illustrated in Figure 4. In an ideal world this would

[1]The word "performance" is used in a general sense.

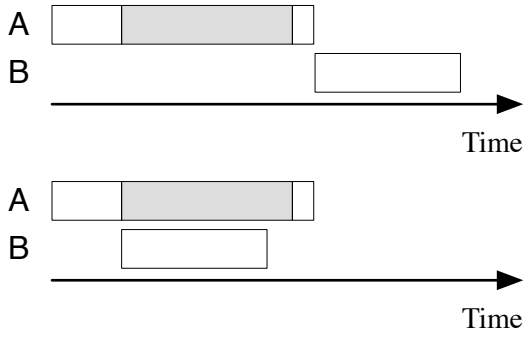

Figure 3: The top diagram shows the execution time in a sequential system that waits during I/O access (grey box). The lower diagram shows how we take advantage of the wait time to run the second process, resulting in better total performance.

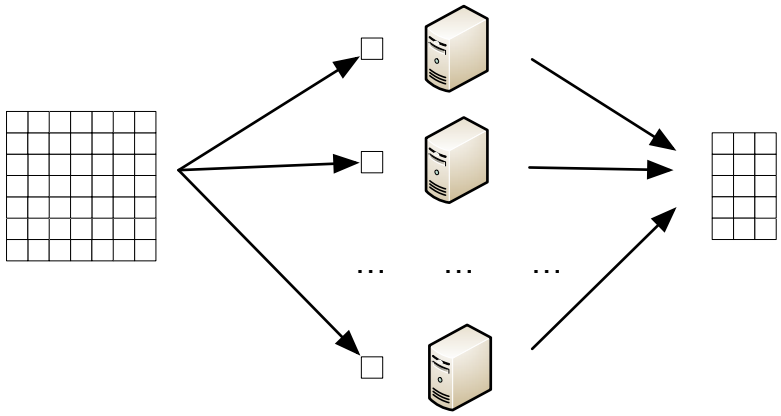

Figure 4: A matrix is divided into several parts, each part is processed on a separate unit and the results are combined.

mean that using ten processors instead of one would cut down the time used to one-tenth of the time.

Both of the examples above involve increased performance, but concurrency can also be used to make it easier for a programmer to design and implement a system. The following fictitious example tries to exemplify this. Let us assume that we have a program that plays video; this means that it needs to read data from a disk or a network connection (R), decode/output sound (DS, OS), decode/output images (DI, OI) in addition to react to user input (UI), see Figure 5. However, it quickly becomes difficult to coordinate these activities so that the system gets new data to process on time, decode/output sound and images without delay/jitter, and at the same time be responsive to user input. By making a design based on concurrency, we can design a system that is much

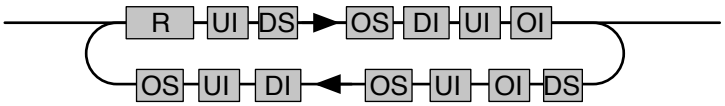

Figure 5: A single threaded solution where many different activities need to be interleaved.

easier to implement and maintain, see Figure 6. Here we see how the different

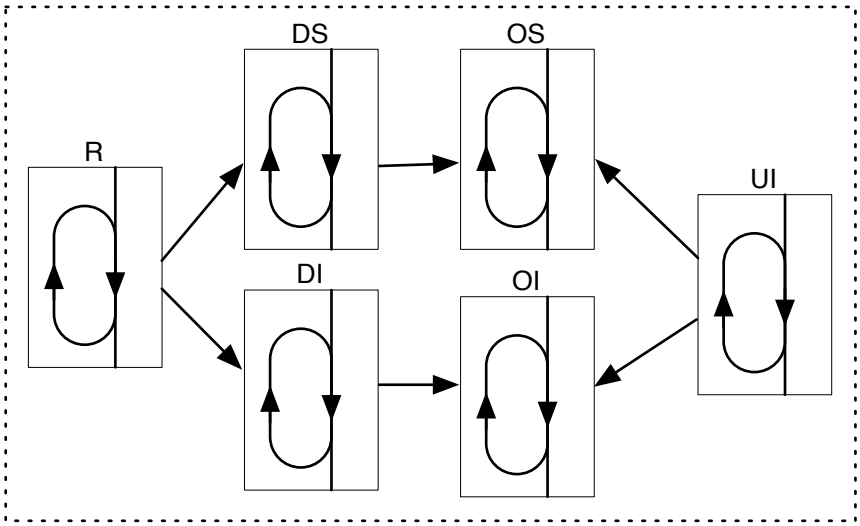

Figure 6: A solution with six threads that communicate to achieve the desired result. Note that this solution is more modular than the one in Figure 5.

threads are responsible for one single task and only communicate with each other when necessary. This solution has two major advantages: designing and implementing the program becomes easier since each task is separated from the others with a clearly defined interaction interface; and in the case where we have multiple processing units, we can schedule the different threads onto different processing units resulting in higher performance of the whole system.

5.3 Problems with concurrency

Unfortunately, concurrency comes with its own set of problems. We will not describe them all here, but only give an example of one of the most basic ones: a race condition. Assume we have two threads, A and B, that execute the following code.

```
Thread A:
A1: while index < 3 do
A2:     sum = sum + values[index]
```

```
A3:     index = index + 1
A4: print sum

Thread B:
B1: if index > 1 then
B2:     index = 0
```

Assume that the value of 'index' is 0 before the execution of this code and that 'values' contains the values [8, 10, 20]. Since these two threads execute concurrently there are several possible sequences in which these instructions can execute. One possibile sequence is A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, A4, B1, B2. This would result in a printout of 38. But an equally possible execution order would be A1, A2, A3, A1, A2, A3, A1, A2, A3, B1, B2, A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, A4 which would result in 76 being printed. The problem that shows in the second sequence is that both threads can freely access 'index' at any time. Thread A executes the loop A1–A3 three times, making 'index' equal three. The next statement would be A1 again, which would evaluate `index < 3` to false and quit the loop. However, before this happens thread B starts and changes the value of 'index' to 0. When thread A now continues with A1, it will evaluate `index < 3` to true and sum up 'values' a second time.

This also means that running the same program twice does not necessarily give the same result. To avoid problems similar to this one, we need some way of synchronizing the access to 'index'. Several synchronization primitives have been suggested and once again we will limit ourselves and describe only one, the semaphore. The semaphore is a very basic synchronization mechanism that can be viewed as a datatype that has the value of 0 or higher. Only two operations are allowed on a semaphore besides initialization: 'acquire' and 'release' – both of these operations are "atomic" meaning that they cannot be interrupted or interleaved. 'Acquire' can be described using the following pseudo-code:

```
if value > 0 then
    decrease value by one
    let the calling thread continue execution with the next instruction
else
    suspend the thread
```

And the 'release' operation:

```
if other threads waiting then
    let one of the waiting threads run
else
    increase value with 1
continue execution of calling thread
```

Given this definition we can avoid the problems described previously by rewriting the code as follows:

```
Init code:
I1: Semaphore indexlock = 1
```

```
Thread A:
A1: acquire(indexlock)
A2: while index < 3 do
A3:     sum = sum + values[index]
A4:     index = index + 1
A5: release(indexlock)
A6: print sum

Thread B:
B1: acquire(indexlock)
B2: if index > 1 then
B3:     index = 0
B4: release(indexlock)
```

This means that we have two possible permutations of the lines I1, A1–A5 and B1-B4: either I1, A1, A2, A3, A4, A5, B1, B2, B3, B4; or I1, B1, B2, B3, B4, A1, A2, A3, A4, A5. When the print statement on line A6 is reached it will print out 38 in both cases.

While this seems to be a simple solution to the problem, there are additional problems that need to be taken into consideration. Take, for example, the case where both threads need resources R1 and R2 to complete their tasks. R1 and R2 can be modeled as semaphores r1 and r2. A possible solution could be written:

```
Thread A:
A1: acquire(r1)
A2: acquire(r2)
A3:   # complete task
A4: release(r2)
A5: release(r1)

Thread B:
B1: acquire(r2)
B2: acquire(r1)
B3:   # complete task
B4: release(r1)
B5: release(r2)
```

Assume the execution order of A1, B1, A2, B3, A3, B3, etc. Already at the third step we can see a potential problem: Thread A tries to get access to r2 which already has been acquired by B. In the fourth step, disaster is complete since B tries to gain access to r1 which is owned by A. In other words, we now have two processes that both are waiting for the other process to release a needed resource, a situation commonly referred to as "deadlock" – the system has come to a complete standstill.

We will not further discuss possible problems that can occur when using a concurrent system; we recommend the interested reader to look at one of the available textbooks for details.

5.4 Problems students are thought to have

Do students have problems with concurrency and if so, what are these problems? We were interested in finding the answers to these questions and turned to the literature [37]. To our surprise we found only a modest number of papers, and many of those we found focused on implementation details of some specific tool. What made this discovery even more interesting was that some of these papers did not describe what problems students have or how the proposed tool would improve the situation. Sometimes we found that the authors of a paper made general claims like "Students have difficulties in learning concurrency since it is difficult to 'see' what happens in the system" or "In our experience students have problem with X". In other words, there were no references to previous studies or empirical data. In some instances a shallow evaluation of some specific tool was described. This work can be summarized as "We showed it to some students. They liked it. You should use it."

However, not everything we found belonged to this category. Several authors have written papers detailing studies investigating student problems and related questions. One author who has written several papers on the subject is Ben-David Kolikant [2, 27, 22, 24, 23, 25, 26, 21]. Some of Ben-David Kolikants findings include:

- Students do not understand what a computational model is.
- Students have difficulties on orchestration synchronization.
- Students tend to make centralized solutions.
- Students have different standards for correctness compared to teachers and professionals.
- The difference in standards for correctness might be attributed to a cultural difference between expert users (the students) and expert programmers (teachers/professionals).

Although it might not seem directly related to concurrency we found the last item especially intriguing. In a concurrent system it is very important that all parts works as specified; if the students have the "relaxed" standards of correctness as described by Ben-David Kolikant, they can easily run into various problems when implementing a concurrent system.

To get a better overview of the problems students face, we classified what different papers claimed to be problematic. This resulted in the following list:

- Concurrency in general
- Previous experience
- Non-determinism
- Tools and programming languages

- Synchronization
- Creating a mental model
- Limited by examples
- Debugging and testing
- Problem solving

Each item is described in more detail in Paper VII. A summary of the papers reviewed is found in Table 5.1, with details found in Paper VII.

We classified the thoroughness of the papers along four dimensions (for a more detailed description of these dimensions see Paper VII):

Claims - Does the paper make any claims about the problems students have? Here we expected that a large percentage of the papers would either belong to "Claims with references" or "No claims". The first group being papers that give some kind of reference to support the claims while the second would contain papers that explore and try to understand student problems. Each paper is placed into one group.

Content - What did the paper discuss? Each paper can be classified as discussing several subjects, this is due to the simple fact that many papers touched on several subjects.

Empirical - Does the paper collect any empirical data? In other words, is data collected for an evaluation of the tool/method/curricula/etc described in the paper? In some cases it makes no sense to collect data since an evaluation make no sense. Each paper is put into one group.

Analysis - Did the paper include any form of data analysis? Please note that there was no judgment on *which* analysis method was used, or of the results. Each paper was put into one group.

While this classification could be analyzed further, it gives an indication of the types of papers we found. We would like to note some interesting information that can be found in Table 5.1:

- As mentioned above we had expected a paper either to make no claims or make claims citing papers with supporting data. To our surprise we found that the group "Claims, no references" was large: 35%.
- Several different subjects are discussed in category "Content"; "Course", "Tool" and "Student understanding" being the ones most popular. We see that only 20 out of 106 papers contain a discussion of student understanding. While this is not in any way conclusive evidence it does give an indication that the interest in "Student understanding" is low.

- To be able to evaluate if/how a tool/course/method/etc affect student learning it is necessary to collect empirical data. Yet, 49% of the papers did not include any mentioning of collecting empirical data.
- We can also see that only 30% of the papers included an analysis of data while 52% did not contain an analysis or an analysis that could be classified as "Anecdotical" or "Questionable".

It should be noted that there are several threats to the validity of these results, mainly in how the papers were collected (see Paper VII for details). On the other hand, we have thoroughly searched the literature using various methods to make sure that we did not miss relevant publications.

Table 5.2 lists how the papers in different content categories differ along the other categories: Claims, Empirical data, and Analysis. Most noticeable is "Student understanding" which has a very different profile than other subjects. Nineteen out of the twenty papers were classified as collecting empirical data and containing an analysis. Ten made no claims about student understanding, while nine that made claims had supporting references. These numbers suggests that papers for this subject can be considered research papers based on empirical data.

Two other subjects worth noting are "Tool" and "Visualization". Both subjects have a large number of papers classified as making claims without supporting data. To a large degree these papers also lack descriptions of collecting empirical data, and few contain an analysis of the data found.

5.5 Problems the students themselves describe

While the problems listed above certainly are interesting they also mainly reflect what teachers/researchers *believe* are the problems students face. By doing a series of interviews, we were able to do a first comparison between what can be found in the literature and what the students themselves think, see Paper VIII.

To highlight some of the results from [38]: Out of the seven students included in the results, six explicitly stated they thought concurrency to be fun/interesting. The seventh student indicated that concurrency was not something he was interested in[2]. This is a very encouraging result; it indicates that the students think that concurrency is something worthwhile to study.

The second thing to note is that the students made very few comments about visualization. In fact, only one student made a comment that we could interpret as a wish for a visualization tool. This is in stark contrast with the literature where visualization is a common theme. There are several possible explanations for this difference, the first being simply that there has been a misunderstanding in what problems students have and what kind of help is needed. But it could also be that the students are not advanced enough to appreciate, or even realize, the difference a visualization tool would make to

[2]although he said that he wanted to work with embedded systems.

Claims	**All**
No claims	47
Claims, no references	37
Claims, references	22
Total	*106*
Content	**All**
Bugs	3
Concurrency concepts	5
Course	40
Language	19
Library	9
Miscellaneous	9
Non-determinism	3
Students understanding	20
Tool	30
Visualization	14
Total	*152*
Empirical	**All**
Anecdotical	23
No	33
Not relevant	15
Yes	35
Total	*106*
Analysis	**All**
Anecdotical	34
Do not discuss learning	6
Yes	32
No	16
Not relevant	13
Questionable	5
Total	*106*

Table 5.1: A summary of articles referenced in Paper VII.

their understanding. Unfortunately, the data collected was not rich enough to allow for a deeper analysis.

Third, is concurrency difficult? Judging by the results from [38], the students do not appear to think so. They seem to think that concurrency is fairly straightforward to understand and use. They do recognize that a concurrent system is more complex than a sequential system, but this added complexity does not appear to deter them from thinking that concurrency is no more difficult than other subjects. In fact, to some the added complexity adds an extra challenge that makes the whole problem solving process more interesting.

	Bugs	Concur.	Course	Lang.	Lib.	Misc.	Non-deter.	Student und.	Tool	Visualiz.
Claims										
No claims		1	20	11	4	6		10	5	
No refer.	2	4	14	8	4	3		1	16	10
References	1		6		1		3	9	9	4
Sum	3	5	40	19	9	9	3	20	30	14
Empirical data										
Anecdotical		2	14	4	2	2		1	5	1
No		2	11	11	4	2			11	5
Not relevant		1	6	2	2	4	1		4	3
Yes	3		9	2	1	1	2	19	10	5
Sum	3	5	40	19	9	9	3	20	30	14
Analysis										
Anecdotical		1	21	5	4	2		1	8	3
Do not disc. learning		1	1	2	1	1	1		1	1
Yes	3		7	2	1	1	2	19	8	4
No		2	4	6	1	3			6	4
Not relevant		1	4	3	1	2			4	1
Questionable			3	1	1				3	1
Sum	3	5	37	14	9	9	3	20	30	14

Table 5.2: Claims, Empirical data, and Analysis for different subjects

There are several sources of errors that might influence these results. The perhaps biggest problem is that we do not know if the students really have understood concurrency. It might be that they *think* that they have understood based on the examples/problems they have encountered during their course and if faced with real-world problems they would change their minds.

Chapter 6

Conclusions

Computer Science is about 60-70 years old (if we exclude early work by Ada Lovelace, etc.) and it is a rapidly evolving field. As a consequence, Computer Science education is chasing a moving target with ever changing programming languages, programming techniques, and tools. However, there are some important concepts that seem to be constant over time, for example, basic concepts like control structures and parameter passing but also more advanced subjects like object-oriented and functional programming.

One of these important concepts is concurrency; it has been important for many decades but we think it is going be even more important in the future. With the increased availability of many-core processors, more and more types of applications are going to use a multi-threaded design.

Encouragingly, students seem to be interested in concurrency and agree the importance of the subject. They also appear to think that concurrency is fairly straightforward to learn. They, of course, have some problems, but students do not indicate that concurrency is radically more difficult than other concepts. On the contrary, the interviewed students seem to think that concurrency is an interesting challenge that they want to master.

This does not mean that concurrency is without its problem. Researchers, for example: Ben-David Kolikant and Lönngård, list several issues that seem to be common problem for novices learning concurrent programming. There are many opportunities for future research about concurrency and learning. We find the questions below especially interesting:

- All students mention at least one synchronization primitive. Some mentioned a specific primitive as being a problem at first – although none said it was a big problem. Was this the first primitive they learned? Were the difficulties "normal" confusion that occurs when a new concept first is introduced? Or, can we see a pattern in what primitives caused problems?

- Have the students really mastered concurrency or are they "stuck" at a lower level of understanding – similar to what is indicated in [29] about programming in general? Have they encountered concurrent systems with

a large number of threads and complex thread interaction? Are the student answers we received only based on experience with systems of basic complexity? Would their answers be different if they had experience from complex systems?

- What kind of software tools are most helpful when trying to learn concurrent programming?
- In Paper VII we find that several different languages have been introduced in attempts to make concurrency more accessible. Is there a combination of language type/notation - for example textual, graphical, object-oriented - and pedagogical approach that yield better results than other combinations.

We believe that by continued research of the problems students experience and how different tools/languages/pedagogical approaches affect them, we will make concurrency easier to learn in the future.

Bibliography

[1] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceeding of the Fourth international Workshop on Computing Education Research*, ICER '08, pages 3–14, New York, NY, USA, 2008. ACM.

[2] Mordechai Ben-Ari and Yifat Ben-David Kolikant. Thinking parallel: the process of learning concurrency. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 13–16, New York, NY, USA, 1999. ACM.

[3] Jeffrey Bonar and Elliot Soloway. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1:133–161, June 1985.

[4] Shirley Booth. *Learning to program: A phenomenographic perspective.* Ph.d. thesis, University of Gothenburg, 1992.

[5] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Threshold concepts in computer science: do they exist and are they useful? In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 504–508, New York, NY, USA, 2007. ACM.

[6] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, third edition edition, 2001.

[7] John W. Creswell. *Qualitative Inquiry and Research Design: Choosing among Five Approaches.* Sage, 2007.

[8] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

[9] Allen Cypher. A stagecast retrospective. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Benedict du Boulay. *Some difficulties of learning to program*, pages 283–299. Lawrence Erlbaum, 1988.

[11] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, Kate Sanders, and Carol Zander. Putting threshold concepts into context in computer science education. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 103–107, New York, NY, USA, 2006. ACM.

[12] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Carol Zander. Can graduating students design software systems? In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 403–407, New York, NY, USA, 2006. ACM.

[13] Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. From limen to lumen: computing students in liminal spaces. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 123–132, New York, NY, USA, 2007. ACM.

[14] Michael W. Eysenck and Mark T. Keane. *Cognitive Psychology: A Student's Handbook*. Lawrence Erlbaum Associates, Publishers, 1990.

[15] Sally Fincher, Josh Tenenberg, and Anthony Robins. Research design: necessary bricolage. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 27–32, New York, NY, USA, 2011. ACM.

[16] Sue Fitzgerald, Beth Simon, and Lynda Thomas. Strategies that students use to trace code: an analysis based in grounded theory. In *ICER '05: Proceedings of the first international workshop on Computing education research*, pages 69–80, New York, NY, USA, 2005. ACM.

[17] T.R.G. Green. *The Nature of Programming*, pages 21–44. Academic Press, 1990.

[18] Sibsankar Haldar and Alex A. Aravind. *Operating Systems*. Pearson Education, 2010.

[19] Ken Kahn. Drawings on napkins, video-game animation, and other ways to program computers. *Communications of the ACM*, 39:49–59, August 1996.

[20] Ken Kahn. Programming by example: generalizing by removing detail. *Communications of the ACM*, 43:104–106, March 2000.

[21] Y. Ben-David Kolikant and M. Mussai. "so my program doesn't run!" definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education*, 18(2):135–151, 2008.

[22] Yifat Ben-David Kolikant. Gardeners and cinema tickets: High school students' preconceptions of concurrency. *Computer Science Education*, 11(3):221 – 245, 2001.

[23] Yifat Ben-David Kolikant. Learning concurrency as an entry point to the community of computer science practitioners. *Journal of Computers in Mathematics and Science Teaching*, 23(1):21–46, 2004.

[24] Yifat Ben-David Kolikant. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies*, 60(2):243–268, February 2004.

[25] Yifat Ben-David Kolikant. Students' alternative standards for correctness. In *Proceedings of the first international workshop on Computing education research*, pages 37–43, New York, NY, USA, 2005. ACM.

[26] Yifat Ben-David Kolikant and Mordechai Ben-Ari. Fertile zones of cultural encounter in computer science education. *Journal of the Learning Sciences*, 17(1):1–32, 2008.

[27] Yifat Ben-David Kolikant, Mordechai Ben-Ari, and Sarah Pollack. The anthropology semaphores. *SIGCSE Bulletin*, 32(3):21–24, 2000.

[28] Gary Lewandowski, Alicia Gutschow, Robert McCartney, Kate Sanders, and Dermot Shinners-Kennedy. What novice programmers don't know. In *Proceedings of the first international workshop on Computing education research*, ICER '05, pages 1–12, New York, NY, USA, 2005. ACM.

[29] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin*, 36:119–150, June 2004.

[30] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees: novice programmers and the solo taxonomy. *SIGCSE Bulletin*, 38:118–122, June 2006.

[31] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceeding of the Fourth international Workshop on Computing Education Research*, ICER '08, pages 101–112, New York, NY, USA, 2008. ACM.

[32] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10:16:1–16:15, November 2010.

[33] Robert McCartney, Jonas Boustedt, Anna Eckerdal, Jan Erik Moström, Kate Sanders, Lynda Thomas, and Carol Zander. Liminal spaces and learning computing. *European Journal of Engineering Education*, 34(4):383–391, 2009.

[34] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. Questions, annotations, and institutions: observations from a study of novice programmers. In Ari Korhonen and Lauri Malmi, editors, *Koli Calling 2004 - Proceedings of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*, volume TKO-A42/04, pages 11–19. Helsinki University of Technology, Department of Computer Science and Engineering, 2004.

[35] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

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

[37] Jan Erik Moström. Learning concurrency – what's the problem? 2011.

[38] Jan Erik Moström. Students experience of learning concurrency. 2011.

[39] Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. Concrete examples of abstraction as manifested in students' transformative experiences. In *ICER '08: Proceeding of the Fourth international Workshop on Computing Education Research*, pages 125–136, New York, NY, USA, 2008. ACM.

[40] Jan Erik Moström, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. Computer science student transformations: changes and causes. In *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, pages 181–185, New York, NY, USA, 2009. ACM.

[41] Brad A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '86, pages 59–66, New York, NY, USA, 1986. ACM.

[42] Brad A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Program. Lang. Syst.*, 12:143–177, April 1990.

[43] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *CHI '06 extended abstracts on Human factors in computing systems*, CHI EA '06, pages 75–80, New York, NY, USA, 2006. ACM.

[44] Marie Nordström. *Object Oriented Quality in Introductory Programming Education*. Ph.d. thesis, Umeå University, 2010.

[45] Arnold N. Pears and Carl Erickson. Enriching online learning resources with "explanograms". In *Proceedings of the 1st international symposium on Information and communication technologies*, ISICT '03, pages 261–266. Trinity College Dublin, 2003.

[46] Vennila Ramalingham and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Empirical Studies of Programmers, Seventh Workshop*, pages 124–139. ACM Press, 1997.

[47] Alexander Repenning and Tamara Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *Computer*, 28:17–25, 1995.

[48] Janet Rountree and Nathan Rountree. Issues regarding threshold concepts in computer science. In *Proceedings of the 11th Australasian Computing Conference (ACE 2009)*, pages 139–145, 2009.

[49] Darrell Patrick Rowbottom. Demystifying threshold concepts. *Journal of Philosophy in Education*, 41(2):263–270, 2007.

[50] Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Lynda Thomas, and Carol Zander. 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, New York, NY, USA, 2008. ACM.

[51] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Common sense computing (episode 4): debugging. *Computer Science Education*, 18(2):117–133, 2008.

[52] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Commonsense computing: what students know before we teach (episode 1: sorting). In *Proceedings of the second international workshop on Computing education research*, ICER '06, pages 29–40, New York, NY, USA, 2006. ACM.

[53] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29:850–858, September 1986.

[54] Elliot Soloway, Kate Ehrlich, and Jeffrey Bonar. Tapping into tacit programming knowledge. In *Proceedings of the 1982 conference on Human factors in computing systems*, CHI '82, pages 52–57, New York, NY, USA, 1982. ACM.

[55] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson Education, seventh edition edition, 2012.

[56] Allison Elliott Tew, W. Michael McCracken, and Mark Guzdial. Impact of alternative introductory courses on programming concept understanding. In *Proceedings of the first international workshop on Computing education research*, ICER '05, pages 25–35, New York, NY, USA, 2005. ACM.

[57] Lynda Thomas, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Kate Sanders, and Carol Zander. *Threshold Concepts in Computer Science: An Ongoing Empirical Investigation*, volume 42 of *EDUCATIONAL FUTURES: RETHINKING THEORY AND PRACTICE*, pages 241–258. Sense Publishers, 2010.

[58] Anne Venables, Grace Tan, and Raymond Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*, ICER '09, pages 117–128, New York, NY, USA, 2009. ACM.

[59] Susan Wiedenbeck. Factors affecting the success of non-majors in learning to program. In *Proceedings of the first international workshop on Computing education research*, ICER '05, pages 13–24, New York, NY, USA, 2005. ACM.

[60] Carol Zander, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Mark Ratcliffe, and Kate Sanders. Threshold concepts in computer science: A multi-national empirical investigation. In Ray Land, H.F Meyer, and Jan Smith, editors, *Threshold Concepts within the disciplines*. Sense Publishers, 2008.

[61] Carol Zander, Jonas Boustedt, Robert McCartney, Jan Erik Moström, Kate Sanders, and Lynda Thomas. Student transformations: are they computer scientists yet? In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 129–140, New York, NY, USA, 2009. ACM.