Educating Novices in The Skills of Programming

Michael E. Caspersen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Educating Novices in The Skills of Programming

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

supported by
it-vest

by
Michael E. Caspersen
February 26, 2007

Abstract

Programming is recognised as one of seven grand challenges in computing education. Decades of research has shown that the major problems novices experience are composition-based —they do not know what the pieces are and do not know how to put them together. Despite this fact, textbooks, educational practice, and programming education research hardly address the issue of teaching the skills needed for systematic development of programs.

We provide a conceptual framework for incremental program development, called *stepwise improvement*, which unifies best practice in modern software development such as test-driven development and refactoring with the prevailing perspective of programming methodology, stepwise refinement. The conceptual framework enables well-defined characterizations of incremental program development; in particular, it enables a notion of degree of correctness, which plays a key role in stepwise improvement.

We utilize the conceptual framework to provide a derived programming methodology for novices and an instructional design for an introductory programming course in which stepwise improvement is supported explicitly by the programming process and a model-driven approach to object-oriented programming and implicitly through cautious design of the teaching material.

Our approach is founded in cognitive science and educational psychology, primarily cognitive load theory, cognitive skill acquisition, and cognitive apprenticeship, as well as research in programming methodology.

Acknowledgments

It is a privilege and a great pleasure to thank the many people who made this dissertation possible.

Gitte Møldrup, director of IT University West, suggested that I pursued a PhD and has provided resources and support for making it possible.

Ole Lehrmann Madsen, my advisor, who urged me for seven years before I gave in, has for more than a decade been a rich source of inspiration and knowledge.

David Gries, my friend, colleague, and long-distance mentor, the latter for almost twenty-five years and not all the time to his knowledge, has offered help and support at a scale far beyond the call of friendship.

Erik Meineche Schmidt, my first teacher in computer science and mentor for many years, taught me programming and how to teach it, and he introduced me to the work of Dijkstra and Gries.

Jens Bennedsen, my close colleague and friend for almost twenty years, has been a wonderful playmate and an endless source of energy throughout our common endeavours.

Ole Eriksen, my colleague and friend for more than twenty years, has been an endless source of inspiration. Many ideas about programming and programming education have matured through countless talks, letters, emails, and cycling tours.

Michael Kölling, my colleague and friend for more than ten years, has made my educational life much easier with his contributions. Through close interaction he continuously influence my way of thinking about teaching programming.

Henrik Bærbak Christensen, my colleague and friend for almost ten years, taught me how to integrate advanced object-oriented concepts into programming education for novices.

Numerous people have contributed to my research. Jens Bennedsen, Henrik Bærbak Christensen, Michael Kölling, Jürgen Börstler, Marie Nordström, Carl Alphonce, Adrienne Decker, and Kasper Dalgaard Larsen are co-authors of one or more of my publications.

Klaus Marius Hansen, Erik Ernst, Aino Corry, Henrik Bærbak Christensen, Carl Alphonce, Adrienne Decker, David Gries, Michael Clarkson, and Michael Schwartzbach offered their time and help as subjects for my study of programming experts in action.

From 2003 through 2006, more than one thousand students from the course *Introduction to Programming* have volunteered as subjects in various experiments and evaluations.

I also take the opportunity to thank Owen Astrachan, David Barnes, Moti Ben-Ari, Joe Bergin, Claus Bossen, Mats Daniels, Anna Eckerdal, Sally Fincher, Gudmund Frandsen, Mark Guzdial, Poul Henriksen, Lew Hitchner, Jens Dybkjær Holbech, Joe Hummel, Michael Jackson, Bruce Klein, Morten Kyng, Bengt Lennartsson, Morten Lindholm, Raymond Lister, Lauri Malmi, Helle Mathiasen, Dung "Zung" Nguyen, Rich Pattis, Eric Roberts, Axel Schmolitzky, Ian Utting, Niklaus Wirth, Stephen Wong, and the members of the Scandinavian Pedagogy of Programming network for occasional interaction and support.

Thanks to all colleagues at DAIMI for providing a wonderful atmosphere of friendship, support, enthusiasm, and professional ambitions.

Last but not least, heartfelt thanks to the most precious in my life, my wife Susanne, my daughter Christina, and my son Christopher; without you it would all be meaningless; for your immense support I shall remain forever grateful.

Thank you all!

Michael E. Caspersen,
Aarhus, 26 February 2007

Contents

I Overview

1 Introduction

Programming education is indeed a grand challenge. Fifty percent failure rates for introductory programming courses are not unusual; study after study have demonstrated that students after six to twelve month of instruction cannot fluently apply basic constructs such as loops; and several studies recognize that even after two years of programming instruction at university, many students have only rudimentary understanding of programming. The community has promoted research and development in the area for thirty-five years without significantly improving the state of affairs.[1]

In 2004 programming education was recognized as one of seven grand challenges in computing education: *the challenge of programming education is to understand the programming process and programmer practice to deliver effective educational transfer of knowledge and skills*.

Our aim is to address the challenge of programming education by capturing the essence of the programming process and programmer practice in order to devise effective education for novices in the skills of programming.

In our endeavour, we shall refrain from discussing the educational inadequacy of modern programming languages. We shall do so not because it is irrelevant —on the contrary— but because it is a separate concern and because our aim is to address the challenge of programming education independently of specific (kinds of) programming languages. Although expressed in terms of object-oriented programming and Java, the dominating teaching language of our time, the essence of our work is applicable to programming education in general, independently of choice of programming language technology.

Programming education, i.e. teaching and learning programming, is the heart of the matter, but teaching and learning are two very different things. Learning is a cognitive, psychological activity that ultimately takes place in the mind of the learner through recoding of cognitive structures; teaching is a social activity in a social system aiming at transferring and constructing knowledge and skills. It is the combination of the two that is our concern and we shall address both aspects of programming education in this dissertation.

1.1 Theses and research questions

As a starting point for this dissertation we propose three theses about teaching and learning introductory programming. The theses are based upon more

[1] This chapter takes the form of an extended abstract without documentation of claims and assertions; everything will be readdressed and documented in the following chapters.

than twenty years of personal experience teaching introductory programming at the tertiary level and have been refined through the PhD study.

> **Thesis 1** (T_1): Revealing the programming process to novices eases and promotes the learning of programming.
>
> **Thesis 2** (T_2): Teaching skills as a supplement to knowledge promotes the learning of programming.
>
> **Thesis 3** (T_3): Anybody can learn to program.

Theses of this nature, of course, are difficult if not impossible to verify, but they are still useful because they form the scope and fundamental perspective of our work. From the three theses, we shall derive the more concrete and specific research questions to be addressed in the dissertation.

As always, it is mandatory to investigate relevant research to look for support or the opposite of the statements put forward in the theses above. The word ‘learning’ occurs in all three theses, and the learning domain is programming education; therefore, the first two research questions to explore are pretty obvious:

> **Research question 1** (Q_1): What is the foundation in learning theory for programming education that supports T_1-T_3? The question is refined to four more specific questions:
>
> $Q_{1.1}$: Is there a foundation in learning theory that supports (or contradicts) T_1-T_3?
>
> $Q_{1.2}$: If so, what are the major relevant results?
>
> $Q_{1.3}$: What is the scientific validity and reliability of these results?
>
> $Q_{1.4}$: Do the results generalise to the context of programming education?
>
> **Research question 2** (Q_2): Does programming education research support T_1-T_3? The question is refined to two more specific questions:
>
> $Q_{2.1}$: What is programming education research all about?
>
> $Q_{2.1}$: Does some of the research support or contradict the claims of T_1-T_3?

Thesis T_3 is a very strong claim, particularly in the light of the current state of affairs as mentioned above. Due to high failure rates in introductory programming courses, it is a common assumption that not everybody can learn to program. This attitude was recently promoted in very strong terms by researchers who claimed to have found a test for programming aptitude to cleanly separate programming sheep from non-programming goats. As succinctly expressed in T_3, we strongly disagree with the attitude that not everybody can learn to program. Of course, we do not mean to suggest that anybody can become a brilliant programmer, but we claim that anybody —provided that they are motivated and that the body of knowledge is suitably structured— can learn the basic knowledge and skills of programming. The thesis is inspired by Jerome Bruner[2] and his attitude toward structure of knowledge. Bruner requests that educators must specify the ways in which a body of knowledge should be structured so that it can be most readily grasped by the learner. Bruner explains it this way: “Any idea or problem or body of knowledge can be presented in a form simple enough so that any particular learner can understand it in a recognisable form” [Bruner 1960]. However, some students learn programming more easily than others and perform much better; it is therefore reasonable to search for explanations, i.e. to search for pre-study as well as in-study indicators of success for introductory programming. To the extent that we as a community are able to identify such indicators, we may be able to use them to improve students’ background and prerequisites to increase their performance and chances of success. This leads us to the third research question to be explored:

> **Research question 3** (Q_3): Are there indicators of success for learning and performance in introductory programming? The question is refined to four more specific questions:
>
> $Q_{3.1}$: Has programming education research identified indicators of success for introductory programming courses?
>
> $Q_{3.2}$: If so, can we generalise the results of others to our local context?
>
> $Q_{3.3}$: Can we identify novel indicators of success in introductory programming?
>
> $Q_{3.4}$: If so, can we exploit these to improve students’ performance and chances of success?

When talking about programming and programming competencies we shall distinguish between *knowledge* and *skills*. Knowledge is facts, definitions, language constructs, specific algorithms, etc., whereas skills are *strategies for using knowledge*. We unfold the definition later, but this shall suffice for now.

Typical introductory programming textbooks devote most of their content to presenting knowledge about a particular language [Robins et al. 2003]. Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming, which appear to deal with ‘program’ as a noun rather than as a verb. But teaching programming is much more than teaching a programming language [Knudsen et al. 1988]. Knowledge about a programming language is a necessary but far from sufficient condition for learning the practice of programming. Students also need knowledge about *the programming process*, i.e. how to *develop* programs, and they need to extend that knowledge into *programming skills* as expressed by theses 1 and 2.

As David Gries once wrote [Gries 1974]:

> *Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Fi-*

[2] Jerome S. Bruner (1915), an American psychologist from Harvard, has made valuable contributions to cognitive psychology and cognitive learning theory in the field of educational psychology known as social constructivism [Bruner 2006].

nally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!

— David Gries, 1974

Clearly, cabinet making cannot be taught simply by teaching the tools of the trade and demonstrating finished products; but neither can computer programming. Nevertheless, judged by the majority of past as well as contemporary textbooks, this is what seems to be attempted. In [Kölling 2003b], a survey of 39 major selling textbooks on introductory programming was presented. The overall conclusion of the survey was that all books are structured according to the language constructs of the programming language; the process of program development is often merely implied rather than explicitly addressed. A typical structure of a section on a specific language construct (e.g. the while loop), is the presentation of a problem followed by a presentation of a program to solve that problem and a discussion of the program's elements. From the viewpoint of a student, the program was developed in a single step, starting from a problem specification and resulting in a working solution. The fact that we all start by developing sub-optimal and partial implementations on our way to a solution, which we later refine and improve, seems to be one of the best kept secrets of programming education! We need to supplement the teaching of tools, concepts, and programming language constructs on the one hand and finished example programs on the other with education in the process of programming. We need to provide *the missing link* between the tools of the trade and products created by these tools, as indicated by Figure 1-1.

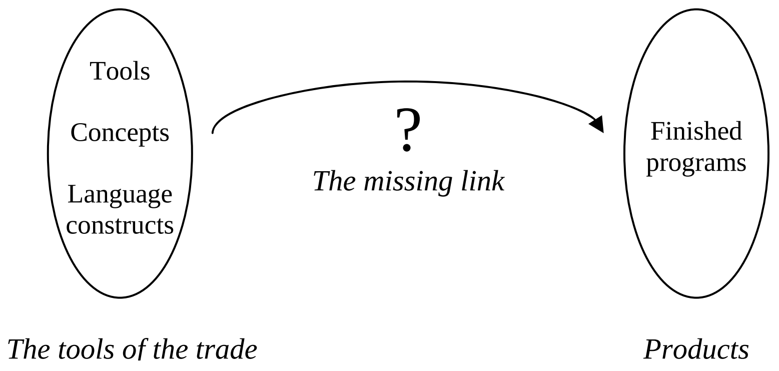


Figure 1-1: *The missing link between the tools of the trade and products*

But in order to incorporate the programming process in programming education for novices, we need to understand what it is. Programming methodology, the study of methods for making programs, aims at increasing programmers' ability to compose programs; theses 1 and 2 are just another way of saying that programming methods ought to permeate programming education in order to ease and promote learning. But where is the field of programming methodology today, and how does it relate to best-practice of modern software development? This becomes our fourth research question:

Research question 4 (Q_4): How does best-practice in modern software development relate to the research area of programming methodology? The question is refined to four more specific questions:

$Q_{4.1}$: How has programming methodology influenced programming education in the past?

$Q_{4.2}$: How can we characterize best-practice of modern software development?

$Q_{4.3}$: How does best-practice in modern software development relate to programming methodology?

$Q_{4.4}$: Can we provide a characterization of the programming process that unifies programming methodology and best-practice of modern software development?

Once we have an understanding of the missing link expressed as a model of the programming process, the follow-up question becomes how to teach this to novices. Consequently, our fifth and last research question is:

Research question 5 (Q_5): How can we educate novices in the skills of programming? The question is refined to four more specific questions:

$Q_{5.1}$: How can we down-scale modern software development methods to a programming process for novices?

$Q_{5.2}$: How can we structure the relevant body of knowledge so that it can be most readily grasped by the learner?

$Q_{5.3}$: How can we organize efficient learning paths/courses that incrementally approximate best-practice in modern software development at the level of novices?

$Q_{5.4}$: How can we adopt results of cognitive science and educational psychology to the instructional design[3] of introductory programming education?

In the dissertation we provide answers for these questions; how and where we do that is exposed in the next section.

1.2 Contributions and organization of the dissertation

The contributions of the PhD study as documented in this dissertation encompass answers to the research questions presented in the previous section.

The first research question: *What is the foundation of learning theory for programming education that supports T_1-T_3?*, is addressed in chapter 3, where we provide a survey of relevant theories and models of cognitive science and educational psychology with focus on their relevance to introductory programming education. This survey is my own and has not been published.

The second research question: *Does programming education research support T_1-T_3?*, is addressed in chapter 4, where we present a map of key activities and publications in programming education research as well as a comprehensive overview of the research field, with special focus on its relevance to my work. This work is my own and has not been published.

[3] Instructional design concerns detailed specification of teaching/instruction as opposed to curriculum design which concerns specifications of learning outcome.

The third research question: *Are there indicators of success for learning and performance in introductory programming?*, is addressed in chapter 5, where we present an overview of related work in the area and three local studies which are described in detail in three papers included in the second part of the dissertation. All of this work was carried out with Bennedsen. Larsen, an undergraduate student, joined us in one of the local studies.

The fourth research question: *How does best-practice in modern software development relate to the research area of programming methodology?*, is addressed in chapter 6 and 7. Chapter 6 addresses question $Q_{4.1}$ and $Q_{4.2}$ by presenting a historical perspective on the role of programming methodology in programming education and providing a brief overview of best-practice of modern software development. Chapter 7 provides answers to $Q_{4.3}$ and $Q_{4.4}$ by exposing the fundamental difference of stepwise refinement and modern techniques of incremental program development and presenting a model of the program process that unifies the two. This work is my own and has not yet been published.

The fifth research question: *How can we educate novices in the skills of programming?*, is addressed in chapter 8 and 9, Chapter 8 provides a brief presentation of a down-scaled programming process for novices. The presentation of the process is structured so that it can most readily be grasped, remembered, and applied by the learner. In chapter 9 we discuss the overall organization of an introductory programming course that incrementally approximates best-practice of modern software development. In doing so, we apply results of cognitive science and educational psychology in general and cognitive load theory in particular to ensure an instructional design of an introductory programming course that balances the cognitive load in order to optimize learning. The work reported in section 8.1.3 was carried out with Bennedsen and is described in detail in the paper in chapter 15. The work reported in section 8.2-0 was carried out with Kölling and is described in detail in the paper in chapter 16. Parts of the work reported in chapter 9 was carried out with Christensen, Bennedsen, Alphonce, and Decker and is described in the chapters 17-21 in the second part of the dissertation. The incorporation of cognitive science and educational psychology is my own work, as is the description of course organization to incrementally approximate best-practice of modern software development; none of this has yet been published.

An important aspect —*perhaps the most important aspect*— of scientific work is the new questions it makes possible to conceive and express. In this light, we consider chapter 10 on future work to be an important contribution of the dissertation, perhaps the most important. With Börstler and Nordström, we have already pursued one of the questions generated from the PhD study; the result of our initial investigations is incorporated as the last of the eleven papers in the second part of the dissertation.

However, before addressing the five research questions we have put forward, we shall elaborate on the challenge of programming education.

2 Programming Education: A Grand Challenge

Programming education is a grand challenge partly because programming intrinsically is hard. To emphasize this point, we begin our endeavour with a broader perspective on programming and a few examples that demonstrate some of the challenges that practitioners experience and education should prepare for.

2.1 Programming is hard

In his speech at the occasion of Edsger W. Dijkstra's 70th birthday and retirement, Niklaus Wirth said:

> *Once he visited us at ETH in Zürich. I was very proud to show to the master my work and explaining some of the difficulties I encountered being well aware that they would probably be difficulties which he understood if he had cared about them, but I was sure he had not.*
>
> *We had one of these rather unfortunate machines, CDC called, with an instruction set now called RISK, but certainly not particularly well suited for our tasks of implementing Algol and higher level languages in an elegant way.*
>
> *I tried to explain to him the difficulties I had been fighting with, and the confessions one had to make, and then —of course with a somewhat haughty look— he said: "Well, I didn't expect anything else from a genuine Swiss puritan". I was a bit puzzled, but I knew enough that I wouldn't take this as exactly a compliment. So I asked him to explain, and he said: "well, the puritans are people who have learned to love their miseries".*
>
> — Niklaus Wirth, 2000

Programming is a creative process; when developing a program for a given problem, programmers are free to invent whatever structures suit their needs, though eventually these structures must be realized in a formal, executable language. It is wise not to worry too early about the final realization of data and processes but instead to invent suitable abstractions for the problem at hand and then —as two separate concerns— solve the problem in terms of the invented abstractions and realize these abstractions on the underlying machine and thereby extend the machine. This strategy can be applied repeatedly and at any level of abstraction.

In principle, the two activities generated when introducing an abstraction are independent in the sense that an abstraction represents a conceptual as well as practical separation of concerns. In practice, of course, the development of suitable abstractions, their use, and their realization on an underlying machine go hand in hand in an incremental, iterative refinement process aiming at optimizing the abstractions according to the requirements and the prevailing criteria of quality.

2.1.1 An explorative activity of discovery and invention

The programming process generated from this strategy —in fact the programming process generated from any strategy— is a process of trial and error; it is *not* a strictly progressive process. If we were always able to pick the right abstractions and get the code right, it would be a strictly progressive process, but we are not. The programming process can be characterized as an explorative activity of discovery and invention where the programmer investigates the problem and the underlying machine, makes hypothesis, and tries to verify these by writing and executing code. In doing so, the programmer creates abstractions, applies these, and realizes the abstractions on the underlying machine.

Ever so often in the process a hypothesis breaks and the programmer must alter or reject it. Either way, it affects the code that has been written, including abstractions that have been created, applied, and realized. Consequently, code must be modified or erased and new hypotheses must be made, from which new code is written and executed.

In order to control the overwhelming complexity that follows from the explorative and non-linear nature of the programming process where hypotheses are altered and the course toward the goal changed, it is vital to base the process on well-defined abstractions. The following parable by Herbert Simon [Simon 1973] explains why:

> *There once were two watchmakers, named Hora and Tempus, who manufactured very fine watches. Both of them were highly regarded, and the phones in their workshops rang frequently. New customers were constantly calling them.*
>
> *However, Hora prospered while Tempus became poorer and poorer and finally lost his shop. What was the reason?*
>
> *The watches the men made consisted of about 1000 parts each. Tempus had so constructed his that if he had one partially assembled and had to put it down --to answer the phone, say-- it immediately fell to pieces and had to be reassembled from the elements. The better the customers liked his watches the more they phoned him and the more difficult it became for him to find enough uninterrupted time to finish a watch.*
>
> *The watches Hora handled were no less complex than those of Tempus, but he had designed them so that he could put together subassemblies of about ten elements each. Ten of these subassemblies, again, could be put together into a larger subassembly and a system of ten of the latter constituted the whole watch. Hence, when Hora had to put down a partly assembled watch in order to answer the phone, he lost only a small part of his work, and he assembled his watches in only a fraction of the man-hours it took Tempus.*
>
> — Herbert A. Simon, 1973

The main reason for advocating this approach is that it is the best (and only) known way of handling the complexity of programming; also, when carried out properly, it produces the most generic solution in the sense of being least dependent on the underlying machine; a third reason for adopting this approach is not to clutter ones thoughts and the program description unnecessarily at too early a stage.

Programming, understood as the process of inventing suitable structures for the problem at hand and, as a separate concern, realizing those as software abstractions of a given software and hardware platform, is notoriously difficult for novices as well as experienced practitioners. We will give two examples of personal experience of the latter.

2.1.2 A tale of two companies

A long time ago, I gave a programming course for programmers in a Danish financial institution. At the introduction on the first day, one group of the participants, the batch programmers, presented themselves with bowed heads and a low voice while the other group, the online programmers (with at least three years of prior experience as batch programmers), proudly and with a loud, clear voice presented themselves. I was puzzled, but being young and inexperienced, I did not realize the magnitude of the distinction. I wanted to understand, so decided to write two COBOL programs, a batch program and an online program, for some trivial computational problem. In doing so, I learned the difference between batch programs and online programs and the reason for the severe distinction between who was allowed to practice what; the batch program took me less than an hour to finish, while the online program took the rest of the day. The point was (and is!) that the runtime environment for online programs[4] does not support I/O: a program that executes an I/O statement is swapped to the disk and restarted in its initial state when input is ready (the program counter and all variables are cleared). Consequently, for every program, application programmers must implement a major part of a process administration algorithm and interleave it with the application code. This is ridiculous and error prone, and it is a waste of the time and mental resources of the programmer —resources that with a proper tool could either have been saved or spent on the real problem. This was the first time I encountered a professional tool that should be categorized as belonging to the problem set rather than to the solution set, and it was an important lesson of the misery that poor tools and lack of education impose on industry.

Many years later, I was involved in a research project collaborating with industrial partners from the domain of embedded, distributed real-time systems; we were researching the applicability of object technology in that domain. In collaboration with a Danish manufacturer of AV-products, we developed a prototype of object-oriented software for an integrated CD player, radio, video recorder, and television. In the existing product developed by the company using traditional software, a real-time operating system (with low-level support for processes, communication, and synchronization) was used to make up for shortcomings of the programming language with respect to support for concurrency. Elements of the real-time operating system were used by so-called task programmers to implement processes, process communication, and process synchronization. However, these aspects of the software was not supported by abstractions but closely interwoven with the

[4] CICS (Customer Information Control System), the runtime environment for online COBOL programs on IBM installations.

application code —in exactly the same way as the process scheduling of online programs was closely interwoven with application code in the previous example.

In both of the previous examples, poor tools, with no direct support for the computational structures of the requirements specification and a conceptual model of the problem domain, caused programmers to struggle unnecessarily to develop application code and —as an integrated part of that— middleware code to make up for shortcomings of their tools. The result is poorly designed software that is hard and, in the long run, almost impossible to maintain. Of course, the chief software architects of the two companies could have developed middleware tools to make up, once and for all, for the shortcomings of the tools. But they did not. Instead they practiced their trade as proud puritans who had learned to (almost) control and therefore love their miseries —the complexity of the tools of their trade.

To the extent practitioners do not invent software abstractions to capture the essential computational structures of the problem and avoid or remove the shortcomings of their tools, I can think of two reasons. The first, and in this context the most important, is poor education. The second is the psychological phenomenon of the prisoner falling in love with his chains; the ability to control complex technology yields respect and admiration from colleagues and peers and generates a feeling of competence, which makes it difficult to realize the fundamental shortcomings of the technology. Eventually, this can also be traced to poor education. It is therefore extremely important to provide excellent programming education to novices as well as experienced programmers. However, providing excellent programming education is not a trivial task either.

2.2 Grand challenges in computing education

In spite of more than forty years of experience, teaching programming is still considered a major challenge; in fact it is considered one of seven grand challenges in computing education. A countless number of sources discuss the difficulties of teaching and learning programming; we will discuss two of those: a review article from 2003 and a report from the British Computer Society on grand challenges for computing education.

In 2003 Robins together and Roundtree published a comprehensive review article on learning and teaching programming [Robins et al. 2003]. In the opening paragraph they write:

> *In recent years the demand for programmers and student interest in programming has grown rapidly, and introductory programming courses have become increasingly popular. Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. It is generally accepted that it takes about 10 years of experience to turn a novice into an expert programmer [...].*

— Robins et al., 2003

2.2.1 Failure rates in introductory programming

In a recent survey of failure rates for introductory programming courses [Bennedsen et al. 2007d], we recently found that the average failure rate in the introductory programming course is 33%. For universities outside the US the result is 41% (Table 2-1). Quite a few major European universities reported failure rates of more than 50%. The maximum failure rate reported was 95%.

	Universities	**Students**	**Avg. fail rate**
Total	54	5,513	33%
Non US	17	2,653	41%

Table 2-1: *Average failure rate for introductory programming courses*

The investigation was carried out by sending direct mail to almost 600 authors of recent CS education research papers. By definition, the target group has a strong concern for CS education and, consequently, is unlikely to be representative for the community as a whole. We expect the average failure rate in general to be even higher than what we found, and we have indications that point in this direction. A colleague from the ACM Education Council [Mark Guzdial. 2006] mentioned an internal report of community colleges (two-year schools) in the US who were in a coalition to improve their retention rates in CS. One school reported an average failure rate, over a ten year period, of 90%! A university with 4000 students, where CS is the second largest major, reported a failure rate of 72%. For comparison, we mention that the failure rates of the past four years of the introductory programming course at the Department of Computer Science at University of Aarhus are 13%, 12%, 12%, and 7% (for a population of approximately 300 students per year).

2.2.2 Grand challenges in computing

In March 2004, the British Computer Society held a conference on the grand challenges in computing [GCC 2004]. GCC ’04 was a collection of three conferences. Two were on grand challenges in computing; these were organised into strands focusing on grand challenges for computing research and grand challenges for computing education. The education strand identified seven grand challenges:

1. *Perception of computing*. Promote an improved and ultimately very positive public image of computing—ensuring that the public gains respect for the field and the professionals who practice within it.
2. *Innovation*. Provide simpler models of computing as a discipline, and have this simplicity reflected in a better mix of high quality computing courses that genuinely accommodate a broad spectrum of student ability and interest.
3. *Competencies*. Ensure that that the quality and currency of computing skills and competence are recognized as important by graduates throughout their career and put in place an infrastructure to provide support and guidance on a career-long basis.

4. *Programming issues*. Understand the programming process and programmer practice to deliver effective educational transfer of knowledge and skills.
5. *Formalism*. Ensure that students of computing see relevant mathematics and formalisms in a very positive light, as providing support, guidance, and illumination.
6. *About e-learning*. Establish e-learning as a credible, viable complement to face-to-face education.
7. *Pre-university issues*. Rationalize the situation at the pre-university level directed toward the promotion of computing to would-be students. Create for students a smooth transition from school to university by enthusing and informing potential students and by creating a positive influence affecting pre-university computing.

One of the seven grand challenges in computing education is 4. Programming issues [McGettrick et al. 2005]. In the report from the conference the authors write:

> *One can generally observe that a strong correlation exists between programming ability and other computing skills, reflecting, as it does, skills in abstraction, conceptualization, design and evaluation. However, major concerns exist among the academic community internationally that when we set out to teach programming skills to students, we are less successful than we need to be and ought to be [...].*
>
> *Given the situation described above, the computing challenge for this area is as follows: Understand the programming process and programmer practice to deliver effective educational transfer of knowledge and skills.*
>
> — McGettrick et al., 2004

Two aspects of the challenge are particularly important: *understand the programming process* and *transfer of skills*. In programming education in general, judged by contemporary textbooks and computing education research, there is hardly any concern for the programming process; the focus seems to be on transferring knowledge rather than skills (for an elaboration, see section 3.5 and 6.1). Invited speaker Peter Denning [Denning 2004a] posed the challenge for us as a field to adopt a new orientation toward programming: as a practice, not a technology. Suppose that programming and problem solving is seen as the essence of our intellectual cachet, and that programmers can have different levels of skill —beginner, advanced beginner, competent, proficient, expert, master— that can be learned through teaching and practice; that effective team skills and customer skills are as essential to programming competence as programming technology; and that software architecture, which concentrates on form and function for good design, is as important as software engineering. Something along these lines is certainly needed if we aim at systematically educating people to design quality software and eventually escape from the miserable state of affairs that was indicated by the two tales in the previous section. As a strategic direction, Denning suggests that a task force examine programming as a core competence of computing, explore what this means, define the levels of competence, and tell us how to teach all this through recommendations for educational objectives and curricula to achieve them.

The question of aptitude for programming was addressed by several participants; Greg Michaelson in his position paper phrased the challenge as follows [Denning et al. 2004]: “If we could somehow characterise the qualities displayed by ‘good’ programming students we might be able to deploy appropriate selection criteria at entry level to better match students to computing courses, find more effective ways of teaching programming both to potential experts and non-experts, and give better support to students who subsequently flounder”. Greg Michaelson proposed a sustained programme of investigation of contributors to and indicators of programming ability including (1) large scale quantitative studies of qualifications and attainments of both successful and unsuccessful programming students; (2) smaller scale longitudinal, cognitive studies of cohorts studying programming in different environments and taught with different approaches; (3) investigation and characterisation of best and worse practice educational approaches to teaching programming.

In another position paper [Denning et al. 2004], Huggard summarises the challenges as how to

a. overcome the negative aptitudes many students have toward programming;
b. establish student aptitude for programming;
c. incorporate the results of psychological studies into our curricula and textbooks;
d. encourage students to practice programming;
e. establish assessment methods that assess individual programming competency effectively;
f. prevent students “hiding” behind their more able peers.

Huggard mentions that individual researchers have sought to address individual issues but concludes that a more coherent approach is needed if we are to radically alter the learning environment in which programming is taught and assessed.

As mentioned in chapter 1, we address challenge a and b in chapter 5. The rest of the dissertation focuses on educating novices in the skills of programming, with particular focus on the programming process and systematic ways of incrementally developing programs and their specifications hand in hand with the latter leading the way. Challenge c is addressed in chapter 3 and 9; challenges d, e, and f are addressed in chapter 9 and 21.

3 Cognition and Learning

A dissertation on educating novices in the skills of programming that does not take the learner into account is of limited value. The purpose of this survey on aspects of cognitive science and educational psychology is to provide a basic conceptual framework for use in the rest of the dissertation when discussing related research and instructional design of programming education for novices. Unfortunately, there is little discussion and research of the teaching of programming that relates to pedagogy, and almost none that address how the process of learning might or should affect instruction [East et al. 1996, p. 1]. The report on strategic directions in computer science education concurs: “We must view changes in pedagogy as opportunistically as we view changes in research specialties” [Tucker 1996]. There is, however, a slow but increasing awareness of the benefits of applying models and research results from cognitive science and learning theory to instructional design and cognitive skill acquisition.

Learning theory is a huge area of which we shall only touch a very small part. The following people (just to mention a few) have all made significant contributions to cognitive science, educational psychology, and learning theory, but we shall refrain from elaborating on their contributions. Still, we may occasionally make references to their work.

- *Jean Piaget* (1896–1980): the theory of cognitive development and (individual) constructivism [Piaget 2007].
- *Benjamin Bloom* (1913–1999): the classification of educational objectives and the theory of mastery learning [Bloom 2007].
- *Jerome Bruner* (1915–): the development of curriculum theory, instructional design, spiral curriculum, and social constructivism (inspired by the work of Lev Vygotsky (1896–1934)) [Bruner 2006].
- *Seymour Papert* (1926–): the theory of constructionism, built upon Piaget’s work on constructivism but went beyond it to assert that learning happens especially well when people are engaged in constructing a product [Papert 2007].
- *John B. Biggs* (?): the SOLO model of constructive alignment in teaching and assessment [Biggs 2003]
- *David A. Kolb* (1939–): the theory of experiential learning and the associated learning model known as Kolb’s learning cycle [Kolb et al. 1975].
- *Howard Gardner* (1943–): the theory of the multiple intelligences [Gardner 1983].
- *Etienne Wenger* (1952–) and *Jean Lave* (1968–): the theory of situated cognition and communities of practice [Lave et al. 1991].
- A*llan Collins, John Brown,* and *S.E. Newman*: the theory of cognitive apprenticeship, which holds that masters of a skill often fail to take into account the implicit processes involved in carrying out complex skills when they are teaching novices [Collins et al. 1991].

The survey starts with a short introduction to the basic models and terminology of cognitive science and educational psychology. Via an interlude on a failed experiment that marked the birth of cognitive load theory, we move on to discuss cognitive load theory, which offers guidance for the design of effective instructional design in general. Finally we turn to the area of worked examples and cognitive skill acquisition (acquiring the ability to solve problems in intellectual tasks). The major sources for the survey are [Atkinson et al. 2000, Clark et al. 2006, Paas et al. 2003, Posner 1993, VanLehn 1996]. Other references appear in the text.

3.1 Cognitive science and educational psychology

We begin by discussing aspects of *human cognitive architecture*. All human learning and work activities rely on two of our memory systems: *working memory* and *long-term memory* and the partnership they share. As its name implies, working memory is the active partner (as you read this and think about its relevance to the dissertation, it is your working memory that does the processing). While in learning mode, new information from the environment is processed in working memory to form knowledge structures called *schemas*, which are stored in long-term memory. Schemas are memory structures that permit us to treat a large number of information elements as if they are a single element. New information entering working memory must be integrated into pre-existing schemas in long-term memory. For this to take place, relevant schemas in long-term memory must be activated and *decoded* into working memory, where integration takes place. The result is an encoding of extended schemas stored in long-term memory. The process is known as *schema acquisition,* and this model of the human cognitive architecture is presented in Figure 3-1. Our model is adopted from Newell, Rosenbloom, and Laird's Soar model described in [Newell et al. 1989]. More detailed models exist, e.g. [Penney 1989], but this one suffices for our purpose.[5]

3.1.1 Schemas

There are two general categories of schemas: schemas that encode *knowledge* and schemas that encode *strategies for using knowledge*; we shall call these *knowledge schemas* and *skill schemas*. We have schemas for all aspects of our cognitive lives. We have knowledge schemas for letters, words, and combinations of words that allow us to read easily and rapidly, and we have skill schemas that allow us to write reports, essays, scientific papers, and dissertations. Schemas for the solution to specific mathematical problems may make us competent at mathematics. And, as programmers, we may have knowledge schemas for programming language constructs and skill schemas for systematically developing a loop from its specification.

[5] Cognitive psychology attempts to present theories of thinking based upon models of the human cognitive architecture. From the point of view of an educator, it is not really important if a cognitive theory truly describes how we think; what is important is the possibility of deriving pedagogically useful insights from the theory [Ben-Ari et al. 2004a].

Schemas are variously named *chunks, plans, templates,* or *idioms*. It is tempting to introduce yet another synonym: *pattern*. However, to avoid confusion, we shall use the term *schema* for cognitive memory structures and reserve the term *pattern* for concrete representations of schemas in a specific domain, e.g. program design (design patterns), algorithm design (elementary patterns and algorithmic patterns), or education (pedagogical patterns). We shall use *chunk* as a general concept for schema, pattern, and any other organization of information or unit of understanding.

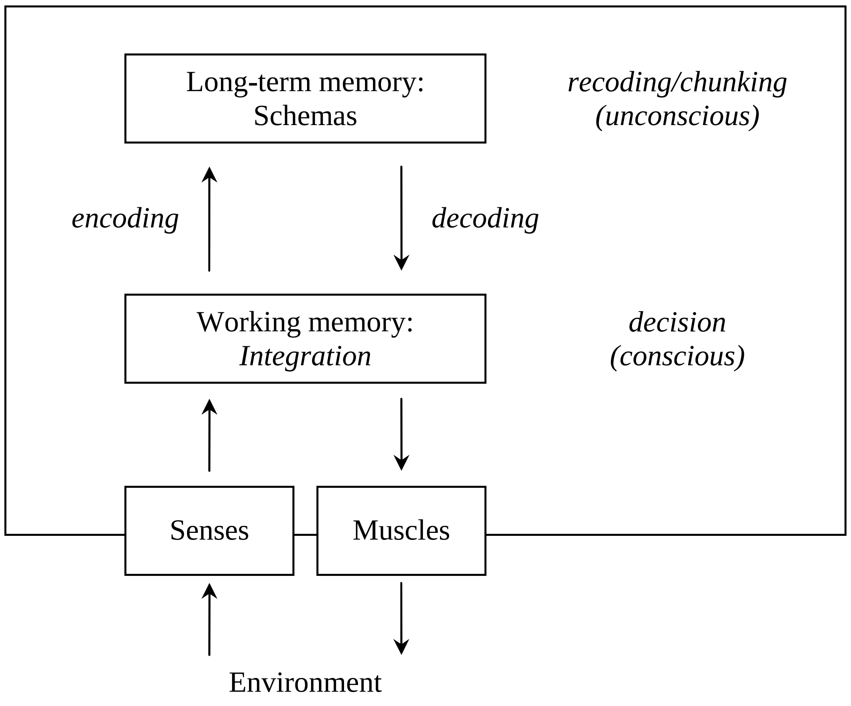

Long-term memory: Schemas
recoding/chunking (unconscious)
encoding
decoding
Working memory: Integration
decision (conscious)
Senses
Muscles
Environment

Figure 3-1: *A model of the human cognitive architecture*

3.1.2 Chunking

In his seminal paper [Miller 1956], George A. Miller observed that the number of chunks of information is constant for working memory. More precisely, Miller found that short-term memory has a capacity of about “seven plus or minus two” chunks —independent of the number of bits per chunk.[6] *Recoding* or *chunking* is the process of reorganizing information from many chunks with few bits of information to fewer chunks of many bits of information. By recoding information, we can make more efficient use of short-term memory and consequently increase the amount of comprehensible information. But there is no free lunch; when recoding we must also learn the associated schemas for decoding/interpretation of information. However, once learned, these schemas are kept in long-term memory and therefore do not affect the cognitive load of short term memory.

[6] Miller’s simple hypothesis is no longer tenable. Chase and Ericsson have showed that purposeful training, based upon metacognitive mnemonic strategies [Allsopp 2007], can triple the apparent working memory capacity [Chase et al. 1981]. However, the fundamental theory of chunking and schema acquisition still applies.

In contrast to working memory, long-term memory has a massive capacity for information storage; however, it is the inert member of the memory partnership. All conscious processing takes place in working memory, but working memory and long-term memory work closely together. The more knowledge and skill schemas stored in long term memory, the greater the virtual capacity of working memory as a result of larger, more complex schemas.

For example, a mid-play chess board includes about twenty-four elements of information for a novice player. However, experts represent the chess board in play patterns involving clusters of several pieces. Each cluster translates into a schema. Therefore a mid-play chess board of twenty-four pieces contains approximately eight or nine schemas for a chess expert. When asked to recall the configuration of the pieces, the experts recalled considerably more of the pieces than did the novices. How would a random placement of pieces on the board affect the recall results? We might expect that experts would lose their advantage and recall about the same number of pieces as novices. Chase and Simon [Chase et al. 1973] tested this hypothesis, and expert player performance was actually worse than novice performance when trying to reconstruct a random configuration! The experts were trying to apply their schemas for chess play patterns to a meaningless environment. The extra psychological work they expended trying to make sense of what they were viewing depressed their memory performance, so they were worse off than the novices. Similar results have been found in programming [Adelson 1981, McKeithen et al. 1981, Shneiderman 1976].

Research on expert-novice differences in problem solving and cognitive skill acquisition indicate that speed and accuracy of experts is not accomplished by major, qualitative changes in their problem solving strategies [VanLehn 1989, p. 563]. The effects of their expertise are more subtle. For instance, whenever an expert and a novice are deciding which chess move to make, both consider the same number of moves and investigate each move for about the same amount of time. The difference is that the expert considers only the good moves and usually chooses the best one, whereas the novice considers mediocre moves as well, and often does not choose the best move from those considered. Thus, expertise lies not in having a more powerful overall strategy or approach but rather in having better knowledge for making decisions at the points where the overall strategy calls for a problem-specific choice. Similarly, experts seem better at monitoring the progress of their problem solving and allocating their efforts appropriately. [Schoenfeld 1981] concludes that *metacognitive or managerial skills* are of paramount importance in human problem solving. The same sort of managerial monitoring is evident in numerous studies including studies of programmers and software design [Jeffries et al. 1981].

3.1.3 Summary

As a result of their enhanced schemas, experts have significantly different psychological capabilities than novices. Experts are able to tackle complex tasks that overwhelm the less experienced. When learning new skills in their domain, experts are enabled by their rich set of schemas to process much larger amounts of information as well as to guide much of their own learning process. Novices, in contrast, lack such schemas and therefore need learning environments that compensate for them. Well-designed learning environments for novices provide *metacognitive managerial guidance* to focus the students' attention and *schema substitutes* by optimizing the limited capacity of working memory in ways that free working memory for learning. Good instruction will segment and sequence the content in ways that reduce the amount of new information novices must process at one time and, as much as possible, reinforce domain patterns to support schema acquisition and improve learning.

3.2 Learning versus problem solving

Cognitive load theory, a special branch of cognitive science and educational psychology, studies how to reduce the load on working memory to optimize learning. The origins of cognitive load theory can be found in the results of a failed experiment [Clark et al. 2006].

3.2.1 A failed experiment

John Sweller and his students were studying transfer effect and *how* people learn while solving problems [Sweller et al. 1982]; in a specific series of experiments it turned out that the students learned nothing! It was a failed experiment with respect to studying transfer, but it sparked the field now known as cognitive load theory.

Seventy-two undergraduate students were given transformation problems in which they were given a start number that had to be transformed into a goal number by finding the right sequence of moves; the only moves permitted were multiplying by 3 or subtracting 69. Problem solvers could use each of these moves as many times as they needed and in any order until they reached the goal number. Each move could be made by pressing a key on a computer keyboard, so no mental arithmetic was involved. At any time, the problem solvers could undo moves. There were many problems, but each problem could be used only by *alternating* the two possible moves a certain number of times. The aim of the experiment was to see what factors would assist problem solvers in learning the rule, but the researchers faced an immediate and inexplicable difficulty. The problems were not very difficult (most people solved them relatively quickly), but despite easily solving the problems, and despite the fact that successfully solving a problem meant applying the alternation rule, very few problem solvers realized the rule! All evidence suggested that most of the problem solvers learnt very little despite being bright university students. It seemed as if learning and problem solving were incompatible! But if learning and problem solving are incompatible, that incompatibility suggested a need for a new view of problem solving, because the field was embarking on a long excursion in which learning through problem solving was a basic assumption —an excursion which also transcended to computer science education in general and programming education in particular.

3.2.2 Misguided advice

Newell and Simon’s book on problem solving [Newell et al. 1972] inaugurated a flowering of the field; for about twenty years, problem solving became one of the central fields of cognitive psychology, and the assumption and advice was that problem solving in one form or another was the best form of learning.[7] While it is now quite clear that this advice was misguided, it is instructive to consider why such erroneous views could become so influential. Proponents of learning through problem solving collected a vast amount of data on the process used by problem solvers, but they *avoided running controlled experiments* in which learning was compared under problem solving as opposed to alternative conditions. Cognitive load theory, the topic of the next section, has proved successful not only because of its reliance on a particular view of human cognition but also because no instructional recommendation generated by the theory has been offered without first being *extensively tested using controlled experiments*.

Back to the failed experiment: The failure of the problem solvers to learn anything useful about the structure of the problems presented to them, led to the obvious question: *How should we have taught them?* In the case of the puzzle problems, the answer is as obvious as the question. If you show people the alternating rule (×3, –69) rather than have them attempt to discover it, they will learn it immediately. As confirmed by both common sense and controlled experiments, while the alternating rule may be hard to discover, it is trivially easy to learn.

Most of the research in cognitive load theory, and in particular the research on worked examples (see section 3.3), was conducted at a time when problem solving practice was a preferred instructional approach, endorsed by many prominent educators and researchers including Newell and Simon as mentioned above. As we shall see in the next section, Sweller's research program accumulated empirical evidence showing that traditional, practice-based problem solving was less than an ideal method for improving problem solving performance when compared to instruction that paired practice problems with worked examples.

3.3 A survey of cognitive load theory

Working memory is the bottleneck of the memory partnership—a very small bottleneck! Cognitive load theory is a set of learning principles that deals with this bottleneck. To be a bit more precise:

Cognitive load is the load on working memory during problem solving, thinking, and reasoning (including perception, memory, language, etc.).

Cognitive load theory is a universal set of learning principles that are proven to result in efficient instructional environments as a consequence of leveraging human cognitive learning processes [Clark et al. 2006].

John Sweller [Sweller 1988] suggests that novices who are unable to recognize a schema to solve a problem must resort to ineffective problem solving strategies like means-ends analysis [Newell et al. 1972]. Sweller suggests that problem solving by means-ends analysis requires a relatively large amount of cognitive processing capacity, which may not be devoted to

[7] It is paramount not to confuse *problem solving* with *problem-based learning*. Problem solving is only what it says, solving problems; problem-based learning is a pedagogical strategy of active learning driven by challenging open-ended problems.

schema construction. Instead of problem solving, Sweller suggests that instructional designers limit cognitive load by designing instructional materials like *worked-examples*, or *goal-free problems*. We return to these later.

The fundamental axiom of cognitive load theory (based upon the model of cognitive architecture) is that learning outcome is optimized when cognitive load fully utilizes the capacity of working memory with elements that allow for optimal schema acquisition. Too little as well as too much cognitive load results in low learning outcome. Routine activities do not advance cognitive development (if there is no new information, no encoding/recoding of schemas take place), and overwhelming with cognitive load does not leave capacity for schema acquisition. Consequently, optimizing learning is a question of *balancing*, not minimizing nor maximizing, cognitive load (see Figure 3-2).

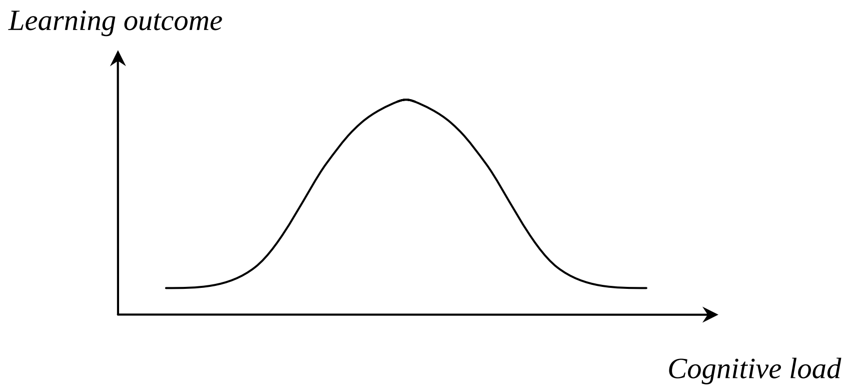


Figure 3-2: *Learning outcome as a function of cognitive load*

However, it is a bit more complicated than that, but also more informative. Cognitive load (*L*) is *currently* divided into three disjoint categories:

- *Extraneous cognitive load* (*E*) is caused by instructional procedures that interfere with, rather than contribute to, learning. Extraneous cognitive load might impede learning, since it requires working memory resources that can no longer be devoted to cognitive processes associated with learning. Furthermore, cognitive resources required by extraneous cognitive load might result in an overall cognitive load that exceeds the limits of working memory capacity.
- *Germane cognitive load* (*G*) is a non-intrinsic cognitive load that contributes to, rather than interferes with, learning by supporting schema acquisition. Germane cognitive load is imposed by adding higher-level cognitive processes that aid schema acquisition and automation.
- *Intrinsic cognitive load* (*I*) is cognitive load intrinsic to the problem that cannot be reduced without reducing understanding. Intrinsic cognitive load depends on the *relational complexity* of the to-be-learned content (so-called *element interactivity*) and the learner's degree of prior knowledge.

Informally, we can express the relationship between *L*, *E*, *G*, and *I* as:

$$L \;=\; E + G + I \quad .$$

In these terms, the challenge of balancing cognitive load for optimal learning becomes a question of minimizing *E* and maximizing *G*.

3.3.1 Milestones in cognitive load theory

Most of the research in cognitive load theory is focused on identifying so-called *effects* (with associated instructional techniques). Initially, researchers in cognitive load theory equated L with E, and all research focused on identifying effects to decrease E. In 1994, cognitive load researchers in the Netherlands identified the *variability effect* (if examples had high variability, cognitive load was increased, but learning was better/more efficient than with low variability examples) [Paas et al. 1994]. This lead to the notion of *germane cognitive load*, hence $L = E + G$. At the same time, researchers in Australia identified the *element-interactivity effect* (some earlier identified effects could be obtained only with material with highly interrelated elements; for sequencable material that could be processed in working memory one or two elements at a time, the effects invariably failed) [Sweller 1994a, Sweller et al. 1994b]. This lead to the notion of *intrinsic cognitive load*, hence $L = E + G + I$. For eight years, the conception was that intrinsic cognitive load is immutable; it could not be varied because it was "intrinsic" to the material. The focus of cognitive load theory was continually to identify effects to reduce E, but now complemented with research to identify effects to increase G. Only in 2002 [Pollock et al. 2002], researchers realized that there had to be ways of reducing intrinsic cognitive load; otherwise very complex material could not be learned. Consequently, the theory was modified to say that I cannot be reduced without reducing understanding. Learning can be facilitated by reducing the number of interacting elements and only reintroduce them later when the essential elements have been learned. However, recent research in cognitive load theory suggests that intrinsic load in favourable circumstances can be reduced without reducing understanding [Gerjets et al. 2004, van Merriënboer et al. 2003]. The milestones are captured in Figure 3-3.

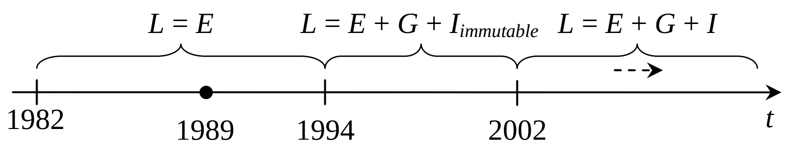


Figure 3-3: *Milestones in the development of cognitive load theory*

The major effects identified so far by cognitive load theory are (in chronological order):

Early years (1982-1994)

- *Lack-of-transfer effect* (1982): The experiment described in section 3.2, marked the birth of cognitive load theory and the $L = E$ assumption [Sweller et al. 1982].
- *Goal-free effect* (1983): Practising through goal-free problems increase learning outcome (an example of a goal free problem in programming is to implement as many methods of a class or system as possible as opposed to telling the students to implement a particular method [Sweller et al. 1983].
- *Worked examples effect* (1985): Alternation of worked examples and problems increase learning outcome and transfer (a worked example is

a demonstration of problem solving by the instructor) [Sweller et al. 1985].

- *Split-attention effect* (1988): Split attention (e.g. combining information in text and diagrams) depress learning. The split-attention effect was demonstrated in worked-examples but applies to all educational material [Chandler et al. 1992, Sweller et al. 1990, Tarmizi et al. 1988, Ward et al. 1990].
- *Refinement of the human cognitive architecture model* (1989): Penney published the paper “Modality effects and the structure of short term verbal memory” which presents an extended model of the human cognitive architecture. The main assumption of the model is that information presented to the auditory and visual modalities is processed in separate working memory streams with independent processing capabilities, i.e. that auditory and visual working memory can be used simultaneously, thus to some extent increasing the capacity of working memory [Penney 1989]. This refinement of the human cognitive architecture was subsequently used to demonstrate the modality effect (1995) and forms with it the basis for recent theories of multimedia learning (2003).
- *Redundancy effect* (1991): Reduce E by reducing redundancy. This effect was rediscovered by many but forgotten because it was counter-intuitive; demonstration of a counter-intuitive effect without a proper theoretical explanation tended to be forgotten and that may be why early examples of the redundancy effect had no impact on the field. Cognitive load theory provides an explanation of the redundancy effect: Most educators intuitively feel that presenting learners with the same information in several different ways cannot be harmful and could be beneficial. But if we have to unnecessarily coordinate multiple sources of the same information, scarce working memory resources are being used for activities unrelated to schema acquisition and automation, depressing learning [Chandler et al. 1991].
- *Example-completion effect* (1992): Rather than giving full worked examples, Paas gave learners completed problems that had to be completed. Partially completed problems presented to learners are as effective as worked examples and better than full problems [Paas 1992]. This effect was subsequently used to demonstrate the guidance-fading effect (2003).

Middle years (1994-2002)

- *Variability effect* (1994): Worked examples with high variability increase cognitive load *and* learning (if intrinsic load is sufficiently low). Identification of this effect lead to the notion of **germane cognitive load**: $L = E + G$ [Paas et al. 1994].
- *Element-interactivity effect* (1994): (some earlier identified effects could be obtained only with material with highly interrelated elements; for sequencable material that could be processed in working memory one or two elements at a time, the effects invariably failed) Identification of this effect lead to the notion of **intrinsic cognitive load**: $L = E + G + I_{immutable}$, where I, as indicated by the subscript designation, is assumed to be immutable [Sweller 1994a, Sweller et al. 1994b].
- *Modality effect* (1995): Mousavi et al. demonstrated this effect under split-attention assumptions (i.e. where two sources of information are

unintelligible in isolation). Controlled experiments comparing geometry diagrams and written or spoken text demonstrated the superiority of the spoken text [Mousavi et al. 1995]. Morano and Mayer extended the result to computer animations with on-screen text and animations [Moreno et al. 1999]. The modality effect, and the extended human cognition architecture model, forms the basis for recent theories of multimedia learning (2003).

- *Expertise-increase, effect-decrease effect of worked examples* (late 1990s): The correlation that effects gradually disappear as students develop expertise was identified in the late 1990s but not published as a result in itself. It was subsequently used to demonstrate the expertise-reversal effect as well as the guidance-fading effect (2003).

Recent years 2002-

- *Isolated-interacting-elements effect* (2002): Pollock et al. realized that there had to be ways of reducing intrinsic cognitive load; otherwise very complex material could not be learned. Consequently, the assumption that I is immutable was abandoned and cognitive load theory was modified to say that I cannot be reduced but only at the expense of a corresponding reduction of understanding [Pollock et al. 2002]. $L = E + G + I$.
- *Expertise-reversal effect* (2003): All previous effects were demonstrated using novices; in the late 1990s, effects were tested under the new conditions of students that had developed some expertise. As mentioned in the expert-increase, effect-decrease effect (1999) effects gradually disappear as students develop expertise. But it turned to be worse than that; it was demonstrated that with further expertise, effects reverse, i.e. the learning outcome was reduced [Kalyuga et al. 2003].
- *Guidance-fading effect* (2003): The expertise-reversal effect was used to demonstrate the guidance-fading effect: complete-examples followed by partially completed examples followed by full problems is superior to any of the three used in isolation [Renkl et al. 2003].
- *Handling large intrinsic cognitive load* (2003-2004): The realization that intrinsic load is not immutable has lead researchers to look for ways to reduce intrinsic load *without* sacrificing understanding. One contribution is a framework for scaffolding practice and just-in-time information presentation to effectively control intrinsic cognitive load [van Merriënboer et al. 2003]. Another contribution represents an instructional technique that applies to problems that can be recursively decomposed to simpler problems that exhibit the same characteristic as the original problem while being simpler [Gerjets et al. 2004].
- *Multimedia learning* (2003-): The demonstration of the modality-effect (1995) and technological development has spawned a new and comprehensive thread of research in cognitive load theory related to multimedia learning [Brünken et al. 2003, Brünken et al. 2004, Mayer et al. 2002, Mayer et al. 2003, Moreno 2004, Schnotz et al. 2005].

Cognitive load theory in general, and the two contributions on *handling large intrinsic cognitive load* in particular, seems highly relevant to programming education; in chapter 9 we discuss application of cognitive load theory for the instructional design of an introductory object-oriented programming course.

Recent developments of cognitive load theory is available via special issues of the journals *Learning and Instruction*, *Educational Psychologist*, *Instructional Science*, and *Educational Technology Research and Development* [Kirschner 2002, Paas et al. 2003, Paas et al. 2004, van Merriënboer et al. 2005]; together the four special issues contain 37 papers (including the four editorials). Furthermore, a textbook is available that presents the theory in accessible form to educators at large [Clark et al. 2006].

3.3.2 Development of cognitive load theory

Researchers in cognitive load theory are careful not to generalise their findings more than controlled experiments allow. An effect that has been demonstrated under some conditions does not necessarily extend to new conditions, and cognitive load theory has developed through a sequence of failed experiments when trying to extend findings to new conditions (e.g. the split-attention effect was identified via failed experiments of extending the worked examples effect; the redundancy effect grew from the split-attention effect in the same manner). Whenever an effect failed to extend to new conditions, cognitive load theory itself was used to explain the failure and provide inspiration for improvement of the theory. In most cases, however, researchers have been able to extend effects to new conditions. In conclusion, we note that a large part of the development of cognitive load theory may be characterised as *direct* or *indirect derivation* of effects (direct meaning successful extension of an effect to new conditions and indirect meaning failure of extension and subsequent *alteration* of the theory).

The anchorage in controlled experiments is good because it means that cognitive load theory represents sound research. However, it is also bad because it means that we cannot as a matter of course claim that results generalise to the context of programming education. But we may anticipate that they do, and it seems more than worthwhile to apply cognitive load theory to the instructional design of programming education; we pursue this in chapter 9.

3.4 Worked examples and cognitive skill acquisition

The survey of cognitive load theory in the previous section is comprehensive, but it represents only a fraction of the amount of research in educational psychology, learning, and pedagogy. As mentioned in the beginning of the chapter we shall not discuss all of this, but we will briefly review the major results of a couple of areas related to and partly overlapping with cognitive load theory. The areas are the *worked examples research* and *cognitive skill acquisition*. The primary resources for this section are two recent review articles: *Learning from Examples. Instructional Principles from the Worked Examples Research* [Atkinson et al. 2000] and *Cognitive Skill Acquisition* [VanLehn 1996].

Studies of students in a variety of instructional situations have shown that students prefer learning from examples rather than learning from other forms of instruction (e.g. [Chi et al. 1989, LeFevre et al. 1986, Pirolli et al. 1985]). Students learn more from studying examples than from solving the same problems themselves [Carroll 1994, Cooper et al. 1987]. A three-year program in algebra was completed in two years by students who studied only examples and solved problems without lectures or other direct instruction [Zhu et al. 1987].

The relevance of the two areas to programming education is explicitly expressed by the authors: “The worked examples literature is particularly relevant to programs of instruction that seek to promote skill acquisition, e.g. music, chess, and programming” [Atkinson et al. 2000]; and “Frequently studied tasks include [...] computer programming” [VanLehn 1996]. Atkinson et al. continues: “[L]earning from worked examples causes learners to develop knowledge structures representing important, early foundations for understanding and using the domain ideas that are illustrated and emphasized by the instructional examples provided. These representations guide problem solving, and they may be conceptualized as representing early stages in domain schema development and in the acquisition of expertise.”

Atkinson et al. have organized their review to emphasize a particular perspective regarding three major categories that influence learning from worked examples; we present the categories as *how-to* principles of constructing and applying examples in education:

1. How to construct examples
2. How to design lessons that include examples
3. How to foster students’ thinking process when studying examples

1. How to construct examples

- *Accentuate subgoals*. Structuring worked examples so that they include cues or beacons that highlight meaningful chunks of information reflecting a problem’s and its solution’s underlying conceptual structure and meaning significantly enhances learning [Catrambone 1998]. Catrambone demonstrates that formatting an example’s solution to accentuate its subgoals can assist a learner in actively inducing the example’s underlying goal structure, and that this cognitive activity presumably helps promote induction of deeper structure representing domain principles, or schemas [Atkinson et al. 2000]. Two techniques have particular efficacy: labels (e.g. verbal specification) and visual separation of steps. Catrambone found that it is the presence of a label, not its semantic content, which influences subgoal formation.

2. How to design lessons that include examples

- *At least add a second example*. Educators must decide how many examples to provide for each problem type. The number may be constrained by external factors, but [Reed and Bolstad 1991] indicates that one example may be insufficient for helping students to induce a usable idea, and that the incorporation of a second example, especially one that is more complex than the first, increases students’ learning outcome significantly. Others have found similar results: education that helps to develop schemas helps in solving problems, and multiple examples of the same schema improves learning and transfer [Gick et al. 1983, Hesketh et al. 1989].
- *Vary form of problem type*. Novices categorize problems according to surface features of the problem statement itself, whereas experts categorize problems according to features and structural similarities of their solution [VanLehn 1989 p. 563]. Variation of form (e.g. cover story) can help novices to realize that there is a many-to-one relation-

ship between form and problem type and vice versa: “when students see the same battery of cover stories used across problem types, they are more likely to notice that surface features are insufficient to distinguish among problem types” [Quilici et al. 1996]. This principle is a supplement to the variability-effect of section 3.3.

- *Alternate examples and practice problems*. Lessons that pair each worked example with a practice problem and intersperse examples throughout practice will produce better outcomes than lessons in which a blocked series of examples is followed by a blocked series of practice problems [Trafton et al. 1993].

3. How to foster students’ thinking process when studying examples

- *Induce self-explanations in example-based instruction*. The message from the large amount of self-explanation literature is clear: students who self-explain outperform students who do not. Furthermore, there are different forms of self-explanation, and students often fail to self-explain successfully; most learners self-explain in a passive and superficial way [Chi et al. 1989, VanLehn 1996]. A good deal of self-explanation research has been conducted in the context of programming education, e.g. [Pirolli 1991, Pirolli et al. 1994].
- *Beware of social incentives*. Social incentives rarely work. Due to the fact that most learners are passive and superficial self-explainers, Researchers have made controlled experiments of initiatives aiming at increasing the quality of degree and quality of self-explanation in various social contexts. In one experiment, students were assigned the role of teacher. The hypothesis was that teaching expectancy would motivate learners to thoroughly self-explain worked examples. In another experiment, students were paired and told to explain examples to each other. Surprisingly, the result of all these experiment was counter-intuitive: Neither teaching expectancy nor peer explanations improved performance; in fact it appeared to hamper learning partly because of increased stress and reduced intrinsic motivation on the part of the students [Atkinson et al. 2000].

Researchers identify three phases of skill development in general, which also apply to cognitive skill development: the *early* phase, the *intermediate* phase, and the *late* phase.

The early phase: In this phase, the subject is trying to understand the domain knowledge without yet trying to apply it. This phase is dominated by reading, discussion, and other general-purpose information acquisition activities. In programming education, however, the image is blurred due to the benefits of technology, which can be used to explore basic domain knowledge by “playing” with the computer (e.g. learn about the development environment and basic language constructs by editing, compiling, and executing simple programs spelled out in the teaching material). In programming education, the early phase includes writing as well as reading, but not problem solving.

The intermediate phase: The intermediate phase begins when students turn their attention to solving problems. When students enter the intermediate phase, they have some relevant knowledge for solving problems but certainly not all of it. There will be *flaws* in the domain knowledge (i.e. misunderstandings and missing knowledge). The purpose of the intermediate phase is to correct flaws and acquire heuristic, experiential knowledge that expedites problem solving [VanLehn 1996]. Eventually, students remove all (or most) flaws in their knowledge and can solve problems without conceptual errors, although they may still make unintended errors, or slips [Norman 1981]. This capability signals the end of the intermediate phase and the beginning of the late phase.

The late phase: During the late phase, students continue to improve in speed and accuracy as they practice, even though their understanding of the domain and their basic approach to solving problems does not change. Practice effects and transfer are the main research issues in this phase.

Of course, this three-phase chronology is an idealization. The boundaries between phases are not as sharp as the description suggests. Moreover, instruction on a cognitive skill is divided into courses, topics, chapters, and sections. Students are introduced to a component of the skill, given substantial practice with it, then moved on to the next component. Thus, at any given time, students may be in the late phase with respect to some components of their skill but in other phases with respect to other components. Nonetheless, it is useful to make the three-phase distinction. [VanLehn 1996].

Hardly any research in cognitive skill development relates to the early phase. Most of what we have discussed so far in this and the previous section relates to the intermediate phase of cognitive skill development. In the remaining part of the section we shall therefore concentrate on the late phase, which is often ignored in higher education. Perhaps the most ubiquitous finding about the late phase of cognitive skill development is *the power law of practice*. Another interesting finding is the effect of *transfer*.

3.4.1 The power law of practice

The power law of practice states that the logarithm of the response time for a particular task decreases linearly with the logarithm of the number of practice trials taken. In other words, response time as a function of number of trials can be expressed as a power function:

$$f(x) = a \cdot x^{\log_2(b)}$$

where a is the response time for the first trial and b is the learning percentage (a learning percentage of 80% means that a doubling of the number of trials will reduce the response time with 20%). A *learning curve* is the graph of a function describing the power law of practice. Figure 3-4 shows the learning curve of a power function with a 50% learning effect and an initial response time of 120 minutes.

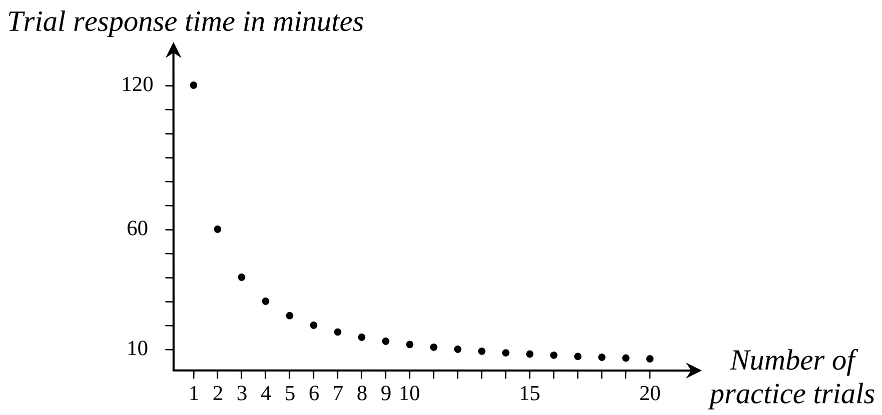


Figure 3-4: *Learning curve with 50% learning effect*

In an influential review [Newell et al. 1981] the authors found that the power law applies to simple cognitive skills as well as perceptual motor skills. Several studies of cognitive skill development found that the speed of applying individual components of knowledge increased according to a power law thus indicating that practice benefits those components rather than the skill as a whole [Anderson et al. 1989, Anderson et al. 1994]. Accuracy also increases according to a power law [Anderson et al. 1994, Logan 1988].

According to [VanLehn 1996], several theories of the power law have been advanced; [Newell 1990, Newell et al. 1981] claimed that chunking of knowledge in long-term memory was the reason, thus allowing the same task to be accomplished by applying fewer schemas. [Anderson 1993] claimed that the speedup is due to two mechanisms: Knowledge in long-term memory is converted from a slow format (declarative format) into a fast format (procedural format), and the speed of individual pieces of procedural knowledge also increase with practice.

The power law of practice is an example of the *learning curve* effect on performance. The learning curve effect and the closely related experience curve effect express the relationship between experience and efficiency. As individuals and/or organizations get more experienced at a task, they usually become more efficient at it. The concepts originate in the adage “practice makes perfect”, and the concepts are opposite to the popular misapprehension that a “steep” learning curve means that something is hard to learn. In fact, a “steep” learning curve implies that something gets easier quickly [Wikipedia 2007b].

Heathcote et al. question the theory that response time is a power function of the number of practice trials; they argue that the evidence for a power law is flawed, because it is based on average data. Heathcote et al. report on a survey that assessed the form of the practice function for individual learners and learning conditions in paradigms that have shaped theories of skill acquisition. They found that the *exponential function* fit better than the power function in all unaveraged data sets. They argue that averaging produced a favour of the power function and conclude: “Clearly, the best candidate for the law of practice is the exponential or APEX function, not the generally accepted power function” [Heathcote et al. 2000]. We have not been able to find further evidence for the claim put forward by Heathcote and his colleagues. However, validity of the claim implies that performance (time and accuracy) of individual learners improves even faster than anticipated with a power law and hence strengthens the argument for practice in favour of increased learning and transfer.

3.4.2 Transfer

Transfer is expressed as a ratio: the time saved in learning the transfer task divided by the time spent learning the training task. One major finding regarding transfer is that the degree of transfer can be predicted by the number of pieces of knowledge shared between the training and transfer tasks (e.g. [Bovair et al. 1990, Singley et al. 1989]). This is a major argument in favour of pattern-based learning (i.e. reinforcement of domain patterns to support schema acquisition and thereby improve learning). The patterns of a domain (e.g. programming) are abstractions of recurring phenomena in that domain or in other words: "pieces of knowledge shared between tasks". Thus, mastering of patterns becomes a guarantee for increased transfer.

As mentioned earlier, practice affects individual pieces of knowledge and not the skill as a whole. When practice on the training task speeds up certain subskills, use of such subskills continues to be fast when used in the transfer task [VanLehn 1996].

3.5 Conclusion

This chapter has offered an overview of elements of learning theory that we consider particularly relevant to our work. Based on a model of the human cognitive architecture, we have provided a survey of cognitive load theory that highlights the major relevant achievements of the quarter of a century in which the area has existed. Along the same lines, we have provided a survey of the research in worked examples and cognitive skill acquisition. Both areas are known and respected for their reliance on empirical research through controlled experiments.

Some of the research on worked examples has been carried out in the area of programming education and has thus demonstrated the applicability of the theories to our field. Overall, we conclude that there are strong indications of a learning-theoretic foundation that supports the two first of the three theses T_1, T_2, and T_3. (T_1: Revealing the programming process to novices eases and promotes the learning of programming. T_2: Teaching skills as a supplement to knowledge promotes the learning of programming. T_3: Anybody can learn to program.)

In chapter 9, we discuss the application of results from cognitive load theory, worked examples, and cognitive skill acquisition to the instructional design of a course aiming at educating novices in the skills of programming.

Whenever relevant, terminology from this chapter will be used throughout the dissertation.

4 Programming Education Research

A vast amount of development and research in computing education is published in journals and proceedings from conferences and workshops. In this section, we provide an overview of the field by describing activities and publications as well as the major research areas. We restrict the overview to issues related to programming education. As we shall see, the prevailing perspective is teaching programming as knowledge; hardly any of the research explicitly addresses the challenge of teaching programming as a skill. Thus, the chapter provides an answer for the second research question:

> **Research question 2** (Q_2): Does programming education research support T_1-T_3? The question is refined to two more specific questions:
>
> $Q_{2.1}$: What is programming education research all about?
>
> $Q_{2.1}$: Does some of the research support or contradict the claims of T_1-T_3?

The first section provides an overview of the field by describing the essential conferences and publications relevant to programming education research. In section 4.2 we present a comprehensive overview of the overwhelming amount of research structured according to ten major research areas.

4.1 Selected conferences and publications

In this section, we provide a brief overview of selected journals, periodicals, conferences, and workshops that relate to programming education (the number in parenthesis marks the year of origin). The selection covers:

- **Journals, Magazines, and Periodicals**
  - *inroads* – The SIGCSE Bulletin (quarterly, 1970)
  - *International Journal of Human-Computer Studies* (empirical studies of programming and software engineering), Elsevier, Academic Press (monthly, 1974)
  - *Computer Science Education* (quarterly, 1990)
  - *Education and Information Technologies*, Kluwer Academic Publishers (bimonthly, 1996)
  - *Journal on Educational Resources in Computing* (JERIC) (quarterly, 2001)
  - *Informatics in Education* (twice a year, 2002)
- **Annual Conferences**
  - SIGCSE: *The Annual Technical Symposium on Computer Science Education* (1970)
  - FIE: *Frontiers in Education* (1971)
  - OOPSLA *Educators' Symposium* (1992)
  - ITiCSE: *The Annual Conference on Innovation and Technology in Computer Science Education* (1996)
  - ACE: Australasian Computing Education Conference (1996)

    - *Koli Calling* (2001)
- **Annual Workshops**
    - PPIG: Psychology of Programming Interest Group (1987)
    - ECOOP (OOPSLA) Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts (1997)
    - Program Visualization Workshop (2000, biannual, held in even years in conjunction with the ITiCSE conference)
    - OOPSLA "Killer Examples" for Design Patterns and Objects First Workshop (2002)
    - ICER: International Computing Education Research Workshop (2005)

4.1.1 SIGCSE

The ACM Special Interest Group in Computer Science Education, SIGCSE, was established in 1968 [SIGCSE 2006]. The first issue of the newsletter, SIGCSE Bulletin, was published in 1969. The first annual technical symposium on computer science education was held in 1970, and the 38th symposium is coming up in March 2007. The symposium, which is hosted in the US, attracts 1200+ attendees every year; the average number of submissions is 300, and the acceptance rate is approximately 33%.

4.1.2 ITiCSE

In 1996, ITiCSE, a conference similar to the SIGCSE technical symposium but hosted in Europe, was initiated to tempt European educators and researches to join the community. The initiative has been a moderate success; ITiCSE attracts approximately 200 attendees every year; the average number of submissions is 200 (from 35 different countries covering every continent excluding Antarctica); the acceptance rate is 30%. A distinguished feature of ITiCSE is the working groups. Every year, four to six working group proposals are accepted, and working group leaders gather researchers with common interests to collaborate on a research topic. The groups initiate their work a couple of months before the conference and convene at the conference to finish their work. A report documenting the work is reviewed and published. Examples of working group topics from the past 11 conferences are: Interacting factors that predict success and failure in a CS1 course; Role and impact of visualization (recurring topic); Study of assessment of programming skills (recurring topic); Research perspectives on the objects-early debate; Assessment; Student cheating; and Teaching introductory programming using LEGO© Mindstorms. Some working groups have produced remarkable research results (in particular the recurring working groups on algorithm visualization and the 2001 and 2004 working groups that conducted multi-institutional studies of programming skills [Lister et al. 2004, McCracken et al. 2001]), and quite a few have been seminal and spawned ongoing research activities and networks.

4.1.3 Koli Calling

In 2001, a group of Finnish researchers in computer science education initiated the Koli Calling conference series [Koli 2006]. The main reason for arranging the conference was to attract interested scholars and educational technologists within the universities in Finland to join their efforts to figure out the future prospects of the field. The conference was also open for fellow participants from Baltic Sea and Nordic countries. Since 2004, the scope of the conference has been broadened to be world-wide, and the best research papers are published in the international journal *Informatics in Education*. The conference and publication in *Informatics in Education* represent a most welcome opening toward the Baltic countries and Eastern Europe.

4.1.4 Workshop in “killer examples” for design patterns

The OOPSLA “Killer Examples” for Design Patterns and Objects First Workshop series was initiated in 2001 [Alphonce 2006]. The Jargon File [Jargon 2003] defines a killer app as an “application that actually makes a sustaining market for a promising but under-utilized technology”. A killer example provides clear and compelling motivation for some concept. The theme of this workshop is killer examples for design patterns and object-oriented concepts. Object orientation is an excellent approach to managing the complexity of large, real-world software systems. Design patterns are an essential part of an object-oriented approach to managing complexity. The purpose of killer examples is to motivate students and pique their curiosity about object orientation and design patterns. The goal of the workshop is to elicit, share, analyze, and critique killer examples from educators and developers from industry. As OOPSLA in general, one of the primary assets of this workshop is that it brings together practitioners, educators, and researchers to achieve the best from all three perspectives.

4.1.5 ICER

Many attendees at education conferences have their primary research interest in other fields of computing but care enough about teaching and education to attend conferences and publish in this area as well. In the second half of the 1990s, an increasing number of “pure” CS education researchers appeared, and the field began to emerge as a discipline in itself [Dale 2002]. The ICER workshop was initiated in 2005 to accommodate the increasing number of researchers with computing education as their primary research area. The ICER workshop is annual; in odd years it is hosted in the US and in even years it is alternately hosted in Europe and Australia. The first European ICER workshop was held in Canterbury in England; the second will be in Aarhus in 2010. From 2008, the doctoral consortium for PhD students in computing science education research will be moved from SIGCSE to ICER, where it really belongs.

4.1.6 Joint Modular Language Conference

The mission of the Joint Modular Languages Conference, which is held every third year, is to explore the concepts of well-structured programming languages and software and those of *teaching good design and programming style*.

4.1.7 Informatics Education Europe (IEE)

The conference series IEE —sponsored by ACM, and the British Computer Society —started in 2006 and aims to bring together Informatics Educators across Europe and to provide a forum for sharing experience, discussing innovative ideas, and identifying common issues to be addressed within

Europe. IEE II (2007) will focus on “Developments in South-East and East Europe” as its main theme. Despite the significant developments in East and South-East European countries in recent years, including the changes in their technological infrastructure and the emergence of competitive software companies, academic staff in the region have not always responded efficiently. The conference will investigate the reasons for this, address problems common among educators related to the development, improvement, delivery and evaluation of Informatics curricula, and provide an opportunity for sharing experiences and best practice examples from all over Europe.

As a supplement to the list of journals, periodicals, conferences, and workshops related to programming education, it is relevant to mention two congregations that address computing education (though in very different ways), the Scandinavian Pedagogy of Programming Network (SPoP), and the ACM Education Board and Council.

4.1.8 Scandinavian Pedagogy of Programming Network (SPoP)

The SPoP network was established in 2004 as a forum for Scandinavian researchers and educators with a special interest in programming education [Bennedsen et al. 2006c]. A considerable amount of interesting work in computing education is being done in Scandinavia, and all who work in the field can benefit from better opportunities of exchange and co-operation. The network is intended as a rather informal, low overhead group that aims at advancing our field. The long term aim is to establish a multi-location, multi-disciplinary group with regular (if infrequent) meetings that helps facilitating computing education projects. This group should include experts from computer science, pedagogic disciplines, psychology, and others. 24 members of the network are currently finishing the book *Reflections on the Teaching of Programming* to be published as a volume of Lecture Notes in Computer Science by Springer-Verlag in 2007 [Bennedsen et al. 2007a].

4.1.9 The ACM Education Board and Council

The general scope of the ACM Education Board is to promote computer science education at all levels and in all ways possible. The Board is an executive-like committee overseeing the Education Council and will initiate, direct, and manage key ACM educational projects. This includes activities such as the promotion of curriculum recommendations, the coordination of educational activities, and efforts to provide educational and information services to the ACM membership.

4.1.10 Personal involvement and commitment

My personal involvement and commitment so far in the community of CS education and CS education research can be summarized as follows:

- Publication in 12 of the above mentioned 17 publications (2000–)
- Member of ACM’s Education Council (2006–)
- Member of the editorial board of Journal of Computer Science Education (2006–)
- Originator and founding member of the Scandinavian Pedagogy of Programming Network (2004)

- Conference chair for ITiCSE 2002 (Aarhus)
- Program committee member for ICER 2007
- Program co-chair for ICER 2008 (Sydney)
- Program co-chair for ICER 2009 (Berkeley)
- Conference chair and program co-chair for ICER 2010 (Aarhus)
- Program committee member for ITiCSE (2000–2002)
- Program committee member for Koli Calling (2002–)
- Program committee member for Joint Modular Language Conference , JMLC (2006–)
- Program committee member for Informatics Education Europe II, (2007–)
- Co-organizer of the OOPSLA “Killer Examples” for Design Patterns and Objects First Workshop (2005–)
- Reviewer for the SIGCSE and ITiCSE conferences (1994–)

4.2 Research areas

Research in computer science education covers a lot of ground. In the book *Computer Science Education Research*, edited by Sally Fincher and Marian Petre, the authors identify ten broad areas that motivate research activities within the field [Fincher et al. 2004]. The areas are not disjoint, but the classification represents a useful map of the territory of topics encountered in CS education research. The ten areas are:

1. Student understanding
2. Animation, visualization, and simulation
3. Teaching methods
4. Assessment
5. Educational technology
6. Transferring professional practice into the classroom
7. Incorporating new developments and new technologies
8. Transferring from campus-based teaching to distance education
9. Recruitment and retention
10. Construction of the discipline

For each of the areas we present a brief and selective overview of research activities that relate to programming education in general and the topic of this dissertation in particular. The research most relevant to this dissertation may be mentioned but will not be discussed in detail in this section; the detailed discussion is postponed to the relevant chapter to allow for a more complete treatment of the research and its relations to, and perhaps implications for, our work.

4.2.1 Student understanding

This area is characterized by investigation of students’ mental and conceptual models, their perceptions and misconceptions. Research in this area is primarily analytical and concerned with understanding *why* students have trouble with a particular topic, concept, or construct and *how* students (or novices) and experts differ in their understanding and perception of things.

A great deal of the research in this area is conceived as multi-disciplinary research in cognitive science and cognitive psychology as well as CS education research. The Psychology of Programming Interest Group (PPIG) [PPIG 2005] arose from ad hoc meetings of researchers and was established as a group in 1987. [Petre et al. 2005] is a special issue of Computer Science Education with a selection of papers from the 2003 and 2004 PPIG workshops.

We organize our overview of this area into five categories: *psychological studies*, *conceptions and misconceptions*, *phenomenographic studies*, *novice behaviour*, and *educator understanding*.

Psychological studies: Psychological studies of programming started in the 1970s but, as Sheil notes, many of the earlier studies of programming were methodologically weak due to poor experiment design, variability of individual programmers, and an underlying naive view of programming skill shaped more by the fashions of contemporary computing practice than by any reasonable appreciation of the complexity of the programming process. The effects reported are small, and yet they are presented as if they establish claims that go far beyond their data [Sheil 1981]. Since then, a number of research psychologists have entered the field and the methodology has improved considerably. [Gilmore 1990] is an example of several papers on experiment design and methodology to be used in programming studies.

In the 1980s, fueled by the growing body of cognitive science, a number of people conducted research in novice programmers' performance. Much of the work focused on identifying the particular problems that students have when learning to program.

In distinguishing between the ability to read and trace programs (program comprehension) and the ability to write or compose programs (program generation), Winslow concludes that "Studies have shown that there is very little correspondence between the ability to write a program and the ability to read one" [Winslow 1996].

Studies by Spohrer and Soloway [Soloway 1986, Soloway et al. 1983, Soloway et al. 1989, Spohrer et al. 1986] and many others conclude that novice programmers know the syntax and semantics of individual statements but do not know how to combine these features into valid programs. In [Spohrer et al. 1986] the authors distinguish between *composition-based problems* (difficulties in putting the pieces together) and *construction-based problems* (misconceptions about language constructs); empirical studies demonstrated that students' difficulties are due to composition-based problems, not construction-based. The authors conclude that "Educators may be able to improve their students' performance by teaching them strategies for putting the pieces of program code together". This result was confirmed by many contemporaries [Linn et al. 1985, Mayer 1981].

Spohrer and Soloway's findings have been confirmed by similar research in the 1990s. The ability to solve a problem requires aptitude beyond the syntax and semantics of a programming language, and errors in students' programs are commonly related to deficiencies in problem solving strategies and insufficient planning, not syntax [Anjaneyulu 1994, Scholtz et al. 1992, Shackelford et al. 1993, Wiedenbeck et al. 1993].

The fact that these problems persist today is documented in more recent studies. The famous ITiCSE 2001 working group [McCracken et al. 2001] investigated first-year student programming proficiency covering 216 students from four universities. Attempting to refine this work, the ITiCSE 2004 working group [Lister et al. 2004] focused on students' capabilities in program comprehension (i.e. to trace programs); this study covered 556 students from 12 universities. A follow-up to the study on program comprehension is documented in [Lister et al. 2006] where Biggs' SOLO taxonomy is used for classification. In another multinational study covering 314 students from 21 institutions in the US, UK, Sweden, and New Zealand, Eckerdal et al. analyzed the design proficiency of graduating computer science students [Eckerdal et al. 2006]. In conclusion, these studies found that students could not generate programs, comprehend programs, or design programs at acceptable levels [Mead et al. 2006].

Robins et al. provides a comprehensive review of the literature relating to the psychological/educational study of programming and reach the following conclusion: "A major recommendation to emerge from the literature is that instruction should focus not only on the learning of new language features, but also on the combination of those features, especially the underlying issue of basic program design. [...] We suggest that programming strategies should receive more and more explicit attention in introductory programming courses. One way to address this would be to introduce many examples of programs as they are being developed (perhaps 'live' in lectures), discussing the strategies used as part of this process" [Robins et al. 2003]. Interestingly, but also sadly, Soloway suggested essentially the same more than twenty years ago [Soloway 1986].

In conclusion: over the past 25 years, study after study, even multi-institutional and multi-national studies, have provided empirical evidence that students cannot program and that the major problems they experience are composition-based —how to put the pieces together. We have a long-standing problem of international scale, which we are aware of, and yet we persist to teach programming primarily by explaining language constructs and show-casing finished programs even though it is procedural knowledge and strategies for putting the pieces together, that is needed!

Conception and misconception: Identifying misconceptions and their causes, and devising ways to address them, constitutes a significant area of science research, and there is a considerable amount of literature on how misconceptions and inappropriate attitudes complicate learning [CUSE 1997]. An interesting perspective on misconceptions is provided by Smith et al. in the paper "Misconceptions Reconceived"; the authors' perspective is that misconceptions are key components of learning and perhaps ought not to be viewed so negatively [Smith et al. 1993].

Clancy provides an overview of research involving misconceptions related to computer programming [Clancy 2004]. The exposition covers background on how misconceptions form, a survey of research that reveals programming-related misconceptions (indicating that the primary reasons for misconceptions are inappropriate transfer/over- and undergeneralization and confusion about computational models). Finally, Clancy discuss ways of dealing with misconceptions and suggests directions for research in the area.

Ragonis and Ben-Ari describe the results of a long-term investigation of the comprehension of object-oriented programming concepts by high school students [Ragonis et al. 2005a, Ragonis et al. 2005b]. The findings were classified as difficulties of misconceptions, and most of them were intermediate, i.e. at the end of the course the students understood the basic principles of object-oriented programming. In total, 58 misconceptions and difficulties were identified and grouped in four categories: object vs. class (17), instantiation and constructors (12), simple vs. composed classes (16), and program flow (13). An example of a misconception is: "After a composed class is defined, new methods cannot be defined in the simple class", and an example of a difficulty is "understanding the influence of method execution on the object state". Of the 58 findings, only 11 were classified as difficulties. Along the same lines, Teif and Hazzan report on a study of high school students' major misconceptions in two more conceptual categories: confusion of (a) taxonomy and (b) partonomic hierarchies with classes, objects, and their interrelations [Teif et al. 2006].

Phenomenographic studies: Phenomenography is a research approach that focuses on the differences in how people experience, perceive and understand a phenomenon [Lister 2003, Marton et al. 1997]. In the early 1970s, early phenomenographers identified two different approaches that students bring to learning. In the "deep" approach, students attempt to develop a genuine understanding of what they are studying, while students using the "surface" approach seek merely to complete the tasks set by the teacher. That phenomenographical research inspired the development of teaching practices for encouraging deep learning [Biggs 2003].

In computer science, Booth conducted the seminal phenomenographic work studying how students experience programming and identified different ways in which students understand recursion [Booth 1997]. More recently, phenomenography has been applied to novices understanding of basic concepts in object-oriented programming [Bruce et al. 2004, Eckerdal et al. 2005a, Eckerdal et al. 2005b, Stamouli et al. 2006]. Currently, Lindholm is conducting a phenomenographic study aiming at understanding how students from the humanities as well as computer science students learn the concepts of 'object', 'class', and 'part-whole relationship' during an introductory programming course [Lindholm 2005, Lindholm 2007].

Student behaviour: Empirical studies of novice programming typically rely on code solutions or test responses as the basis of their analyses. While such data can provide insight into novice programming knowledge, they say little about the programming processes in which novices engage [Hundhausen et al. 2006]. For those interested in improving novice programming environments, a key research question arises: How can we collect and analyze data on novice programming that will enable us (a) to analyze and compare the programming processes promoted by alternative novice programming environments and (b) ultimately to build better novice programming environments?

In order to address this question, Hundhausen et al. have collected a large video corpus of novices as they construct code solutions. Through detailed post-hoc analyses of the video corpus, the authors have developed a methodology for compiling the moment-by-moment evolution of novice code solutions. Based on an analysis of a model code solution's key semantic components, the methodology enables researchers to document, on a second-by-second basis, (a) what part of a code solution a programmer is focusing on and (b) where the semantic feedback provided by the programming environment is helping. Although it is time and labour intensive, the methodology provides researchers with a standard set of data and representations for comparing the programming processes promoted by alternative programming environments [Hundhausen et al. 2006].

Jadud has explored what he calls *compilation behaviour* of novices, i.e. the programming behaviour a student engages in while repeatedly editing and compiling programs [Jadud 2006]. Based on a sequence of compilation events from a student's programming session, Jadud defines the so-called *error quotient* (EQ) which is a number between 0.0 and 1.0 that characterizes how much or how little a student struggles with compilation errors while programming. Jadud found a distinct correlation between a student's EQ and both average grades received on assignments and the final exam". While there is a correlation between EQ and performance, Jadud concludes that he cannot make any strong claims about whether a first-year student's compilation behaviour can be used as a predictor for the performance on traditional exam-based metrics.

Educator understanding: Although this topic clearly falls outside the scope of this section, it is interesting to note that a few researchers recently, or even currently, are extending some of the studies made with students as subjects to educators. Bennedsen and Schulte have conducted a quantitative study of educators' perception of 'objects-first' [Bennedsen et al. 2007f], and Thompson is currently conducting a phenomenographic study of perceptions of object-oriented software development by academics teaching object-orientation and experienced practitioners' [Thompson 2006].

4.2.2 Animation, visualization, and simulation

This area covers research that uses software tools to enhance student learning. Animation and visualization of algorithms and data structures is dominating the area (e.g. [Naps 2006]); related topics are visualization of recursion and visualization of variables and their roles. Computer architecture visualization and network simulations of differing situations and conditions are other popular topics, and so is visualization of parallelism and computational models (e.g. Turing machines, finite automata, and formal languages) [Ben-Ari 2006a, Ben-Ari 2006b, Ibbett et al. 2006, Rodger 2006b, Rodger et al. 2006a]. The biannual Program Visualization Workshop, which in even years is held in conjunction with the ITiCSE conference, is a fruitful activity for researchers in this area [PVW 2000, PVW 2002, PVW 2004, PVW 2006] as is the recurring working groups at the ITiCSE conferences:

- 1996: An overview of visualization: its use and design: report of the working group in visualization [Bergin et al. 1996]
- 1997: Using the WWW as the delivery mechanism for interactive, visualization-based instructional modules [Naps et al. 1997]
- 2002: Exploring the role of visualization and engagement in computer science education [Naps et al. 2002]
- 2003: Evaluating the educational impact of visualization [Naps et al. 2003]
- 2006: Merging interactive visualizations with hypertextbooks and course management [Rössling et al. 2006]

More general activities are the ACM Symposium on Software Visualization [SoftVis 2006] and workshops at other major conferences (e.g. OOPSLA and ICSE workshops on software visualization).

Algorithm visualization systems research dominated the field until the mid-1990s, and systems research fell primarily into two main efforts: expanding the communicative expressiveness of the visualizations and facilitating the creation of the visualizations [Stasko et al. 2004]. In particular, the technique of *interactive prediction* was introduced to allow students to become interactively involved in animations and visualizations. In the mid-1990s, researchers' attention turned toward pedagogical effectiveness of animation and visualization, and a body of empirical research was conducted using distinct empirical methods such as *controlled experiments*, *observational studies*, *questionnaires and surveys*, *ethnographic field techniques*, and *usability studies*. In 2002, Hundhausen, Douglas, and Stasko made a meta-study of the effectiveness of algorithm visualization (AV) where they present a comprehensive review of 24 experimental studies of controlled experiments, including a classification and analysis with respect to underlying learning theories [Hundhausen et al. 2002]. The general conclusion of the meta-study is that "*how* students use AV technology, rather that *what* students see, appears to have the greatest impact on educational effectiveness. [...] In particular, our meta-study suggests that the most successful use of AV technology are those in which the technology is used as a vehicle for actively engaging students in the process of learning algorithms [...] in such activities as what-if analyses of algorithmic behaviour, prediction exercises, and programming exercises [...]. Rather than being an instrument for the transfer of knowledge, AV technology serves as a catalyst for learning".

The positive affective effect of using program visualization, which are typically claimed by the developers of such systems, has recently been documented [Ebel et al. 2006].

A comprehensive and interesting research project on *roles of variables* (variable patterns) [Ben-Ari et al. 2004b, Sajaniemi 2006] has produced 35 publications in the past five years and includes the development of *PlanAni*, a role-based program animator [Sajaniemi et al. 2003].

4.2.3 Teaching methods

This is a very broad area which we shall break down into six sub-areas: *learner-centered education*, *constructivism*, *case studies and apprenticeship*, *the inverted curriculum*, *patterns*, and *teaching and learning object-oriented concepts*.

Learner-centered education: In a special issue of Communications of the ACM, Norman and Spohrer wrote a short editorial, which begins: "A revolution is taking place in education, one that deals with the philosophy of how one teaches, of the relationship between teacher and student, of the way in which a classroom is structured, and the nature of curriculum" [Norman et al. 1996]. The basic issues can be described through such keywords as *constructivism*, *learner-centered*, and *problem-based*. At the heart is the idea that people learn best when engrossed in the topic, motivated to seek out new knowledge and skills because they need them in order to solve the problem at hand. The goal is active exploration, construction, and learning rather than the passivity of lecture attendance and textbook reading.

There is an important difference between learning through problem solving (section 3.2) and problem-based learning. From their designation, the two are easy to mix or equate, but if we take a closer look, they are obviously very different. In the past, the focus has been on the content; curriculum is structured around the basic topics of the content area, and experts divide the topics into small, manageable bundles taught according to a prescribed lesson plan, and the learners practice by solving problems—the more problems that are solved, the better. The new learner-centered approach is somewhat akin to the "user-centered" focus of modern interface design. Here, the focus is on the needs, skills, and interests of the learner. Learner-centered is often accompanied by a problem-based approach, where the problems are picked to fit the interest and needs of the learners. The focus is on the learner and authentic problems rather than on the structured analysis of the curriculum content though both are clearly necessary.

In the paper *How Much Choice is Too Much?* [Becker 2006], the author writes: "Providing a learner-centered perspective is in keeping with modern constructivist approaches to learning, and this means that courses must be designed with learner attributes and choice in mind. Concerns over accreditation and the need for accountability at the post-secondary level seem to contradict freedom of choice and flexibility of term work, but this need not be the case." Becker outlines numerous strategies for offering choice and flexibility to students in a freshman programming course. Approaches include flexible deadlines, the ability to re-submit work that has already been assessed, writing tasks, contributing to course content, bonuses for embellishments and extra work, and choices about which problems to solve. All of the strategies have been employed in classes, and students' reactions as well as effects on student engagement and quality of work are described in the paper. Along the same lines, Bergin et al. report on a study on an examination of the effect of self-regulated learning in introductory programming performance. The results of the study indicate that self-regulated learning is important in learning how to program and can be used to partially predict performance in the course. The study represents initial investigations into the area that warrants further research [Bergin et al. 2005b].

Constructivism: Several researchers and educators in CS have addressed learner-centered education through the notion of constructivism, i.e. that knowledge is constructed from experience: "Learning takes place through the active behaviour of the student: it is what he does that he learns, not what the teacher does", [Tyler 1949]. Basically, there are two schools of constructivism: the Piaget school of individual (or cognitive) constructivism and the Vygotsky-school of social constructivism. Ben-Ari has addressed both in two separate papers [Ben-Ari 2001, Ben-Ari 2004]. Hadjerrouit discuss a constructivist approach to object-oriented design and programming [Hadjerrouit 1999] and constructivism as a guiding philosophy for software engineering education [Hadjerrouit 2005]. A more elaborate discussion of constructivism can be found in [Phillips 1995].

Case studies and apprenticeship: In the 1990s, several related tracks were followed in an attempt to find more effective ways of motivating and presenting material in introductory programming courses [Kölling et al. 2004]. Seminal among these attempts was Pattis' paper *A Philosophy and Example of CS-1 Programming Projects* [Pattis 1990] and the work of Linn and Clancy [Linn et al. 1992], who made a strong argument for the use of case studies to support program design. Particularly effective in their study was the use of expert commentary to accompany a design. Also significant is the work of Astrachan and his colleagues on an applied apprentice approach to CS1 and CS2, which encourage students to read, study, modify, and extend programs written by experienced programmers [Astrachan et al. 1995, Astrachan et al. 1997].

One of the principles of the apprentice-based approach to programming education is that particular applications are the motivation for introducing new programming constructs or data structures, rather than studying constructs and facts about algorithms and data structures as ends in themselves. Kölling and Barnes have written a textbook according to this pedagogical principle [Barnes et al. 2006]. In [Kölling et al. 2004], they describe in detail the rationale, motivation, and goals of the problem-based approach of the book. In the problem-driven approach, it is not the lecture content that drives the assignment, but the assignment problems that drive the lectures. Also, the problem-driven approach makes it possible to achieve the inclusion of modern software engineering tasks into the computing curriculum early on. Traditionally, early computing assignments often use a blank screen approach: students start with nothing more than a problem specification. They then start designing and coding a new application from scratch. The essential assignment task is to write code. This style does not reflect realities in the contemporary computing industry, where tasks like reading and understanding of existing code, maintenance and refactoring, adaptation and extension are far more common than the development of new applications. Following Barnes and Kölling's approach, first students *observe* the teacher demonstrating and extending an existing piece of software using new techniques or constructs introduced for the purpose, then students *apply* the new material to the project under guidance, and finally the students *design* their own tasks as extensions of the project at hand. In this way, the order of student activities is exactly reversed compared to classical, clean-slate assignments; there, students typically have to start with design, followed by applying new material before they observe behaviour.

All of these initiatives correspond to the pedagogical theory of cognitive apprenticeship [Collins et al. 1989] and Wenger's theory of situated cognition and communities of practice [Lave et al. 1991] (see also the introduction to chapter 3).

The inverted curriculum: A radical proposal—at least considering the time when it was put forward— along the lines of postponing the teaching of facts until they are actually needed, is Reek's paper on a so-called top-down approach to teaching introductory programming focusing on understanding the abstractions represented by the classical data structures without regard to their physical implementation; only after the students are comfortable with the behaviour and applications of the major data structures do they learn about their implementations or the basic data types like arrays and pointers that are used [Reek 1995].

Reek's proposal is an incarnation of Meyer's idea of the inverted curriculum [Meyer 1993]. However, Reek was not the first to implement this approach; [Decker et al. 1993] describe a similar approach. More recent incarnations of this idea are [Roumani 2006] and [Pedroni et al. 2006].

In [Buck et al. 2000], Buck and Stucki claim that "traditional approaches to CS1 and CS2 are not in congruence with cognitive learning theory" and provide arguments for a reversed order of topics based on Bloom's classification of educational objectives. The title of the paper is "Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development", and the message of the paper is that the ordering of topics that best matches Bloom's hierarchy of cognitive development is the reverse of the order of activities in the classical software lifecycle model. The authors first do implementation of methods within an existing design, then later they move to design, analysis, and requirements in later courses. "This is counter-intuitive, because it is not the order we work in when we develop systems" [Buck et al. 2000]. The approach is characterized as "teaching software development from the inside out rather than beginning with either console apps or monolithic designs.

Patterns (for schema creation): The fundamental motivation for a pattern-based approach to teaching programming is that patterns capture chunks of programming knowledge; according to cognitive science and educational psychology, explicit teaching of patterns reinforces schema acquisition as long as the total cognitive load is "controlled".

Over the past ten years, a number of computer science educators have begun to incorporate software patterns into their undergraduate courses [Wallingford 2000]. Ideas similar to patterns can be traced back to the work of Mayer [Mayer 1981], Soloway [Soloway 1986], Rist [Rist 1989], and Linn and Clancy [Linn et al. 1992], but consideration of patterns accelerated following the appearance of the pioneering book Design Patterns [Gamma 1995]. The so-called "Gang-of-Four" book documented patterns of object-oriented (OO) design at a time when many CS educators were struggling to master the discipline. Encouraged by the benefits they realized from studying design patterns, some educators began to teach design patterns in their OO courses. The design pattern aspect of teaching patterns is covered in greater detail in section 4.2.6 *Transferring practice into the Classroom*.

East [East et al. 1996] and Wallingford [Wallingford 1996] were among the first to accept the challenge put forward by Soloway in 1986, but soon after, others followed [Astrachan et al. 1997, Astrachan et al. 1998, Reed 1998].

A group of people have developed a catalogue of so-called elementary patterns particularly intended for novices [Bergin 2006a].

Some have pursued the idea of patterns to capture pedagogical knowledge and experience in computer science education in general [Bergin 2006b] and object-oriented programming education in particular [Eckstein 2001]. Computer Science Education recently had a special issue on pedagogic patterns [Fincher 2006] with two local contributions [Bennedsen 2006e, Bennedsen et al. 2006f]. Fincher and Utting discuss potential improvements of pedagogical patterns [Fincher et al. 2002].

We shall return to this issue and discuss it with more focus and in greater detail in section 6.1.4.

Teaching and learning of object-oriented concepts: The special area of teaching and learning object-oriented concepts attracts much attention. The OOSPLA Educators' Symposium and the ECOOP (OOPSLA) Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts were mentioned in section 0. The SIGCSE and ITiCSE conferences always have at least a session or two on the topic. In 2003, Computer Science Education devoted a special issue to the topic [Börstler et al. 2003]. Recently, Sicilia provided a discussion of strategies for teaching object-oriented concepts with Java [Sicilia 2006]. However, these are just the tip of the iceberg; many of the references in the other sections in this chapter could have been categorized under this label as well.

4.2.4 Assessment

We break this broad area into three sub-areas: *assessment methods*, *automated assessment*, and *validity of assessment*. Assessment may be formative (conducted *during* learning to promote, not merely judge or grade, student success), or summative (administered *after* learning is supposed to have occurred to determine whether it did), and it may address different kinds of learning such as acquisition of factual knowledge, change in conceptual understanding, or acquisition of skills [Fincher et al. 2004].

Assessment methods: There are many alternative assessment methods. [Stiggins 2005] enumerates four main categories: *selected response* (multiple choice questions and short answer questions), *essay* (poster presentation, written report), *performance assessment* (case study, practicum, project, and reflective journal/diary), and *personal communication* (class presentation, interview, and learning contract). A typical oral examination is classified as personal communication (interview), and a written examination is classified as selected response (short answer question). According to [Stiggins 2005], each category has advantages in assessing different learning outcomes. For example, a selected response assessment task, such as a series of multiple-choice questions, is able to assess all areas of mastery of knowledge but only some kinds of reasoning.

Multiple choice tests have recently been coming into favour as a useful evaluation tool at the university level [Brown 2001, Roberts 2006, Woodford et al. 2005], in contrast to the earlier view that they support only superficial learning [Biggs 2003]. A recent result regarding multiple choice is that five self-evident axioms are sufficient to determine completely the unique correct scoring strategy for multiple choice tests where students are allowed to check several boxes to convey partial knowledge [Frandsen et al. 2006].

Based upon Bloom's classification of educational objectives [Bloom et al. 1956], Lister and Leaney have developed a criterion-referenced grading scheme to cope with diversity among students in an introductory programming class. In the traditional norm-referencing approach to grading, all students attempt the same programming tasks, and the attempts are graded "to a curve". The danger is that such tasks are aimed at a hypothetical average student. Weaker students can do few of these tasks and learn little. Meanwhile, these tasks do not stretch the stronger students, so they too are denied an opportunity to learn: "Our contribution has been to bring disparate grading techniques together, uniting them in a coherent grading philosophy" [Lister et al. 2003].

Automated assessment: There are at least three motives for conducting automated assessment in programming courses in which the outcome of an assessment is one or several programs or program fragments. One motivation is to reduce the workload of grading assignments, another is to detect plagiarism. The third motivation is that it is difficult to visually inspect program source code and determine whether it is syntactically and dynamically correct, let alone whether it will meet a given specification. Automated assessment systems can be classified according to the notions of formative and summative introduced above; formative automated assessment systems are also known as intelligent tutoring systems. Summative automated assessment systems can be bipartited according to whether they test program comprehension or program generation.

[Brusilovsky et al. 2005a] is an example of a system that assesses program comprehension. Edwards is an example of an inclusive system that assesses program generation based upon a test-driven approach to programming [Edwards 2003a, Edwards 2003b]. The system provides concrete feedback about which portions of the program require (further) testing and about coding style. [Kumar 2005b]is an example of an intelligent tutoring system that can automatically generate problems, answers, grades, and feedback. The system is targeted at a programming language course, but the author is currently developing a suite of automated tutors for an introductory CS course; the tutors are adaptive i.e. the tutors generate problems tailored to the learning needs of the student. A group of systems use evaluation-based approaches to assess student knowledge about algorithms and data structures; the systems TRAKLA and TRAKLA2 [Malmi et al. 2005], which are specifically designed to assess algorithm simulation exercises and allow students to resubmit their solutions after obtaining feedback, are among the best known in this category. Two course designs applying the TRAKLA systems are described in [Malmi et al. 2007].

Automated assessment is covered by two recent reviews; Ala-Mutka provides a survey of automated assessment approaches for programming assignments [Ala-Mutka 2005], and JERIC (Journal of Educational Resources in Computing) recently had a special issue on automated assessment of programming assignments [Brusilovsky et al. 2005b]. From this special issue we particularly recommend the review paper by Douce et al. on automatic test-based assessment of programming [Douce et al. 2005].

Plagiarism is a major issue in CS education as exposed by an ITiCSE 2002 working group [Dick et al. 2002]. Several assessment systems incorporate plagiarism detection; [Lancaster et al. 2004] provides a comparison of source code plagiarism detection engines. Daly and Horgan present a system based on watermarks, allowing them to distinguish supplier and recipient [Daly et al. 2005].

As in the case of algorithm visualization, systems research has dominated the field of automated assessment systems so far, but some evaluations are beginning to appear [Karavirta et al. 2006, Korhonen et al. 2002, Kumar 2005a, Malmi et al. 2005, Traynor et al. 2006]. However, a challenge yet to be met is to conduct meta-studies that compare different systems.

Validity of assessment: Finally, some research is aimed at understanding whether the assessment is valid, i.e. whether it represents the kind of knowledge the educator wants it to assess [Fincher et al. 2004]. An example of this concerns assessment of early programming competence. With reference to the findings of [McCracken et al. 2001], which shows that many computing students are not able to develop straightforward programs after the introductory programming sequence, [Daly et al. 2004] argue that normal student assessment should have highlighted this problem; since it did not, normal assessment of programming ability does not work. The authors examine why current assessment methods (written exams and programming assignments) are faulty and investigate another method of assessment: the lab exam. Furthermore, the authors show that this form of assessment is more accurate, and they explain why accurate assessment is essential in order to encourage students to develop programming ability. This is of particular relevance to our work because we have adopted a lab exam for our introductory programming course; we discuss this issue in section 9.2.8 and in chapter 21.

4.2.5 Educational technology

A number of integrated development environments (IDEs), micro worlds, and tools have been developed to support various aspects of program development and programming for novices.

Integrated development environments: The BlueJ system [Kölling et al. 2003] is the best known educational development environment for Java with excellent support for unit testing [Patterson et al. 2003]. BlueJ started as the Blue project aiming at developing a programming language [Kölling et al. 1996a] and an associated development environment [Kölling et al. 1996b] for teaching object-oriented programming to novices. The development environment is particularly strong in its support for dynamic object inspection and interaction [Rosenberg et al. 1997]. With the appearance and prevalence of Java, the *Blue* project teamed up with Sun Microsystems in a joint effort at providing a pedagogical development environment for Java. BlueJ became the name of the environment [Nourie 2002], and with that, development of an object-oriented teaching language ceased. There are a number of evaluations that report on the use of BlueJ in introductory programming courses [Haaster et al. 2004, Hegna et al. 2006, Ragonis et al. 2005a].

Another widely used Java environment is DrJava [Allen et al. 2002], which is a Java variant of DrScheme [Findler et al. 2002]. The important feature about DrJava is its interaction pane, which allows any expression or statement to be evaluated or executed immediately. This allows for simple demonstrations and facilitates the delay of requiring a method main to the latter part of the course. Yet another educational IDE for Java is jGRASP [Cross et al. 2006] but it has a very limited dissemination. Recently, the BlueJ team and Sun Microsystems announced the Netbeans IDE 5.0 BlueJ Edition, which provides seamless transition from BlueJ to the professional Netbeans development environment [Netbeans 2006].

Micro worlds: The best known micro world for programming is probably *LOGO* developed by Papert and others at MIT in 1967 [Papert 1993]. *Karel J Robot* is the best known and most widely used micro-world for object-oriented programming [Bergin 2007, Bergin et al. 2005]. *Alice* is another example of a well-known micro world for object-oriented programming [Cooper et al. 2003]. By programming within the context of a micro world, students can work with a rich set of primitive objects to illustrate the key concepts of object-oriented languages.

Tools: Other types of tools are being developed to support programming education. Greenfoot is an interesting novel tool, which is a framework that allows easy construction of micro worlds implemented in Java; users/students can extend the behaviour of the micro world by writing Java code that defines the behaviour of objects in the micro world [Henriksen et al. 2004, Kölling et al. 2005]. Greenfoot is developed by members of the BlueJ team and builds upon the principles of direct interaction from BlueJ. Greenfoot is targeted at programming education at the secondary level but has potential for being used at the tertiary level as well.

Java Power Tools (JPT) is a Java toolkit designed to enable students to rapidly develop graphical user interfaces in freshman computer science programming projects. Because it is simple to create GUIs using JPT, students can focus on the more fundamental issues of computer science rather than on widget management. Also, JPT can help freshman students to learn about the basics of algorithms, data structures, classes, and interface design [Proulx et al. 2002, Rasala et al. 2001].

UML[8] class diagrams are used quite commonly in introductory object-oriented programming courses, and some tools have been developed or adopted for this use. Green is a simple, flexible, and extensible UML class diagramming tool provided as a plug-in for Eclipse, which allows students to alternate between class and code view of their projects [Alphonce et al. 2005]. Ideogramic UML is a tool for gesture-based collaborative modeling with UML, which can be used to collaboratively teach and learn modeling [Hansen et al. 2002].

Webworlds is a characterization of web-based learning environments created for programming and software engineering education; many of the specific tools in this category are animation and visualization tools. A somehow dated overview of webworlds for learning software engineering is provided by [Chalk 2000]. A more recent and very comprehensive webworld is Kumar's online tutor system, *Problets* [Kumar 2007]. Problets are problem solving software assistants for learning, reinforcement and assessment of programming concepts. They are designed to help students learn programming concepts through small-scale problem-solving, and as a supplement to large-scale programming traditionally used in introductory programming courses. The Problets have been extensively evaluated, and there is evidence that the use of Problets help students learn: "Students who use the tutor for practice learn better than those who use a printed workbook. Students who receive both graphic visualization and text explanation learn better than those who receive only graphic visualization. Students who use graphic visualization learn better than those who receive no explanation" [Kumar 2005a].

A special sub-area is the use of new technology in the classroom, e.g. tablet PCs. Examples of initiatives of this kind is presented in [Anderson et al. 2004, Denning et al. 2006, Wilkerson et al. 2005]. Koile and Singer report on two pilot studies that evaluate the use of Tablet PCs and a Tablet-PC-based classroom presentation system in an introductory computer science class; the preliminary results seem to indicate that the use of educational technology increase the students' performance [Koile et al. 2006].

A more detailed overview of the environments literature can be found in [Guzdial 2004]; it is suggested by Guzdial that "the greatest contributions to be made in this field is not in building yet more novice programming envi-

[8] Unified Modeling Language [UML 2007].

ronments but in figuring out how to study the ones we have". Gross and Powers have just started to undertake that challenge [Gross et al. 2005].

4.2.6 Transferring practice into the classroom

Computer science is a vocational discipline, which means that a large group of professionals are developing and expanding the practices of the discipline, in parallel with academia. One research strand takes as its focus the transfer of professional practice into the classroom [Fincher et al. 2004]. Examples of recent major contributions to the programming practices primarily offered by people outside academia are *design patterns and frameworks*, *extreme programming*, *refactoring*, *agile development*, and *test-driven development* [Beck 1999, Beck 2003, Cockburn 2002, Fowler 1999, Gamma 1995, Martin 2003].

Design patterns have made their way into the standard curriculum as new courses on software architecture [Christensen 2004]. A recurring OOPSLA workshop, recruiting participants from industry as well as academia, is devoted to discussions on how to best teach design patterns [Alphonce 2006] (see section 4.1.4); highlights from the workshop series are covered in the paper included as chapter 20. Several others have addressed the issue of teaching design patterns in the introductory courses [Astrachan et al. 1998, Clancy et al. 1999, Gelfand et al. 1998, Nguyen et al. 1999, Preiss 1999]. Some even argue that the objects-first approach to introductory programming courses should be modified into a design patterns first approach [Pecinovský et al. 2006]. In chapter 18 we argue that frameworks should be introduced early to train programmers to become software re-users as much as software producers and as a conceptual way of addressing event-driven programming and the associated design patterns; the chapter also illustrate the pedagogical principle *consume before produce*[9] by applying minimal frameworks to build understanding (schemas) and thereby provide stepping stones toward real and more complex frameworks (e.g. GUI frameworks).

Extreme programming typically manifests itself in the classroom as pair programming [Bergin et al. 2004, Williams et al. 2001].

Agile software development in education is covered by a special issue of Computer Science Education [Williams et al. 2002]; practical software engineering education was the topic for another special issue of the same journal in 2001 [Saiedian 2001].

Several educators promote rethinking of the introductory programming course in terms of test-driven programming [Edwards 2004, Janzen et al. 2006, Jones 2004].

4.2.7 Incorporating new developments and new technologies

Almost by definition, this area is the most ephemeral of CS education research. However, not all topics that fall in this category are gone tomorrow.

[9] The principle *consume before produce* is described in section 9.1.1.

The use of robots in introductory programming courses is very popular, and the development of Lego Mindstorm has generated quite a few initiatives as documented by an ITiCSE 2002 working group [Lawhead et al. 2002] and several papers [Blank et al. 2003, Harlan et al. 2001, Imberman et al. 2005]. Fagin and Merkle provide a comprehensive study among more than 800 students (including a control group) of the effectiveness of robots in teaching computer science [Fagin et al. 2003].

In the late 1990s, several researchers and educators argued that introductory computer science education was entrenched in an outdated computational model of *computation as calculation* and that the model ought to be altered to *computation as interaction*; computation as it occurs in spreadsheets, video games, web applications, and robots [Stein 1998]. In the wake of this movement, course designs and textbooks encompassing the *computation as interaction* model were developed [Bruce et al. 2001, Bruce et al. 2005, Hansen et al. 2004, Stein 2003].

Some topics are conceived as new, although the fundamental principles are old; thought-provoking examples of this are presented in [Ben-Ari 2006c, Standage 1998].

4.2.8 Transferring to distance education

Like all other disciplines, CS is increasingly taught as distance education or elearning; this holds particularly for part-time education. Most programming education publications in this area report on transformations of courses from face to face education to elearning. Edwards reports from a workshop on “Establishing a Distance Education Program” [Edwards et al. 2000], and Gersting reports on computer science distance education experiences in Hawaii [Gersting 2000]. Bennedsen and Caspersen describe a web-based introductory programming course for adults and the rationale for choosing technologies and organizing the course in order to provide flexibility and compensate for the drawbacks inherent in this kind of teaching [Bennedsen et al. 2003].

An early and special pioneer in this field is the Runestone project [Daniels et al. 1999]. Within Runestone, students work on a computer science project in teams, under academic supervision. The especially interesting feature of Runestone is that half the students in each team are from Sweden and half are from the US [Last et al. 2002]. They live and work in different time zones and never meet face to face. Yet they work on the same project, for which they are assessed as in any other similar piece of academic work [Fincher et al. 2004].

4.2.9 Recruitment and retention

Due to the recent large decline in enrolments, this area has currently more focus than normal.

One of the big issues in this area is the search for success indicators of programming aptitude. However, we postpone the discussion of the issue to chapter 5, where our own research in the area is presented.

Other topics of interest in this area are attrition rates, diversity, and gender issues. Beaubouef and Mason investigate the possible causes for high attrition rates for computer science students [Beaubouef et al. 2005]. Cuny and Aspray report from a workshop that convened a group of experts to discuss the recruitment and retention of women in computer science and engineering graduate programs [Cuny et al. 2002]. Fisher and Margolis conducted a four-year study of gender issues in the undergraduate computer science program at Carnegie Mellon University; the significant and colourful findings, and the subsequent dramatic increase in the number of women in the program (from 7% in 1995 to 42% in 2000), are reported in [Margolis et al. 2002].

At Georgia Tech, Guzdial and his colleagues have developed a first course in computer science based on media computation and aimed especially at non-majors; of 121 students taking the course, 2/3 were female students. Data from interviews with women from the class is provided in [Rich et al. 2004].

4.2.10 Constructing the discipline

This category is of a different kind than the previous, concerning questions about the construction of the discipline of computer science education and, derived from that, computer science education research.

We don't have the same consensus as, say, mathematics or physics about what the basic concepts are and how to teach them. In some domains, for example mathematics, there is a *didactics*, a sense of what should be taught, an acknowledgement of what fundamental principles should be covered, and an associated understanding of which curricular areas are advanced and which are optional. There can also be a sense of how subjects should be taught. We clearly do not have agreement on that in computer science, although the ACM computing curricula [ACM 2005] and, in particular, the computing curricula CC2001 [ACM 2001] have generated a real discourse around these areas.

ACM's Education Board (recently reorganized to Board and Council) was established in order to develop curriculum recommendations and has worked on this for almost 40 years resulting in five recommendations: *Curriculum '68* [Atchison et al. 1968], *Curriculum 78* [Austing et al. 1979], *Computing Curricula 1991* [Turner 1991], *Computing Curricula 2001* [ACM 2001], and *Computing Curricula 2005* [ACM 2005]. *Computing Curricula 2005* represents a broadening of the area to cover computer engineering, information systems, software engineering, and associate degree programs along with traditional computer science, and *computing* is becoming the general concept that covers all of the more specific areas. Currently, ACM's Education Council has formed four task forces addressing accreditation, curriculum recommendations, enrolment, and technology & tools.

As well as curriculum construction, the area encompasses questions concerning the nature of the discipline: Is it an engineering discipline? Is it mathematics? Is it design? Is it business? And this leads to discussions of interpretations and scope; of how many things the discipline actually embraces [Fincher et al. 2004]. A contemporary example of this is Denning's column in Communications of the ACM on the profession of IT [Denning 2001, Denning 2002, Denning 2003, Denning 2004b, Denning et al. 2005]. Important examples from the last couple of decades are [Comer et al. 1989, Dahlbom et al. 1997, Denning et al. 1997, Dijkstra 1989, Dijkstra et al. 1989, GCC 2004, Simons et al. 1991, Wegner et al. 1996].

Strategic directions in computer science education are outlined in [Tucker 1996]. One of the elements of the strategy concerns research in curriculum and teaching methodologies: "it is imperative that faculty remain actively engaged in the discipline via a continuing scholarly activity [...], this might mean conducting research in "non-traditional" areas such as curriculum and teaching methodologies themselves. Institutions that strongly emphasize teaching must moderate their teaching loads and increase their real support for scholarly activities such as these". As far as we know, this is the first mention of a strategic initiative in computer science education research. The more recent conference on grand challenges in computing education has already been mentioned [McGettrick et al. 2005].

In [Clancy et al. 2001], five senior people from the community present their perspective on CS education research; the paper is a suite of five short papers, which aim to provide an overview of several aspects of CS education research, especially: previous work of interest, current projects and results, suggestions and resources for getting started in CS education research, and for forming and entering research communities. The book *Computer Science Education Research* [Fincher et al. 2004] was a natural follow up to [Clancy et al. 2001] with contributions from more or less the same group of people. This book is a valuable contribution to the construction of the discipline of computer science education research—or computing education research which, in the light of the general broadening of the discipline, perhaps is a more appropriate designation.

4.3 Conclusion

We have presented a comprehensive overview of the overwhelming amount of programming education research by describing key conferences and publications relevant to the community and by capturing the essence of the research in programming education structured according to ten major research areas. Apart from providing the overview, which is worthwhile in itself, the primary motivation for the chapter was to answer the second of our research questions:

> **Research question 2** (Q_2): Does programming education research support T_1-T_3? The question is refined to two more specific questions:
>
> $Q_{2.1}$: What is programming education research all about?
>
> $Q_{2.1}$: Does some of the research support or contradict the claims of T_1-T_3?

For convenience, we repeat the three theses from section 1.1. T_1: Revealing the programming process to novices eases and promotes the learning of programming. T_2: Teaching skills as a supplement to knowledge promotes the learning of programming. T_3: Anybody can learn to program.

The most striking indirect support for T_1 and T_2 in the huge body of research in programming education is the research on student understanding reported in section 4.2.1. The conclusion is clear and unambiguous: students struggle to learn programming, they are not very successful, and the major problem they experience is not syntax but how to put the pieces together.

The research on case studies and apprenticeship and the pattern-based approach to teaching programming described in section 4.2.3 on teaching methods aim at teaching the skill of programming; as such it also supports our perspective, and our work is closely related to and inspired by these approaches. However, except for a few authors who mention live programming in class as a teaching method, virtually no research address the challenge of revealing, let alone teaching, the programming process to novices.

In conclusion: to the extent that research addresses the issues of teaching skills and process, it supports theses T_1 and T_2. We pursue this issue in chapter 6, 7, 8, and 9.

There is no support of T_3 in the literature, on the contrary! The general theme of much research, and the specific results of some, is that (teaching) programming is hard and that students learn much less than we expect from them. A thorough discussion of this issue follows in the next chapter.

5 Programming Aptitude

Thesis T_3 of this dissertation is that anybody can learn to program. As mentioned in chapter 1, we do not mean to suggest that anybody can become a brilliant programmer; we claim that anybody —provided that they are motivated and that the body of knowledge is suitably structured— can learn the basic knowledge and skills of programming. Even this interpretation is perhaps overly optimistic, maybe even naive, but so be it. The purpose of thesis 3 is not to claim a universal truth but rather to present the fundamental perspective of our approach to students and to programming education. There are of course exceptions, but in general we believe that anybody can learn to program provided the material is presented in a suitable form and that the student wants to learn.

However, while it may be true that anybody can learn to program, it is evident that some students learn programming much easier than others and perform much better; it is therefore reasonable to search for explanations, i.e. to search for indicators of success in introductory programming and thereby approach an understanding of what constitutes programming aptitude. To the extent that we as a community are able to identify such indicators, we may be able to use them to improve students' background and prerequisites to increase their performance and chances of success. This perspective is our motivation for doing research in this area.

Understanding programming aptitude was established as one of the aspects of the grand challenge of programming education recognized by the GCC '04 conference on grand challenges in computing [GCC 2004]. Greg Michaelson in his position paper phrased the challenge as follows [Denning et al. 2004]: "If we could somehow characterise the qualities displayed by 'good' programming students we might be able to deploy appropriate selection criteria at entry level to better match students to computing courses, find more effective ways of teaching programming both to potential experts and non-experts, and give better support to students who subsequently flounder". Greg Michaelson proposes a sustained programme of investigation of contributors to and indicators of programming ability including (1) large scale quantitative studies of qualifications and attainments of both successful and unsuccessful programming students; (2) smaller scale longitudinal, cognitive studies of cohorts studying programming in different environments and taught with different approaches; (3) investigation and characterisation of best and worse practice educational approaches to teaching programming.

This has lead to the formulation of the following research questions which are addressed in this chapter:

> **Research question 3** (Q_3): Are there indicators of success for learning and performance in introductory programming? The question is refined to four more specific questions:
>
> $Q_{3.1}$: Has programming education research identified indicators of success for introductory programming courses?

$Q_{3.2}$: If so, can we generalise the results of others to our local context?

$Q_{3.3}$: Can we identify novel indicators of success in introductory programming?

$Q_{3.4}$: If so, can we exploit these to improve students' performance and chances of success?

We shall answer these questions by first providing an overview of research and known results in this area (section 5.1). In section 5.2, we briefly describe our initial investigations of eight potential success factors of which only two was found to be significant: the math grade from high school and student effort. Realizing that it hardly is specific math competencies such as calculus, geometry, or trigonometry that have a positive impact on programming performance, we hypothesised that it is the more general notions of abstraction and reasoning ability that indicates programming aptitude. Our investigations of this hypothesis are described in section 5.3. In late 2005, colleagues in U.K. announced a "scientific breakthrough"; they claimed to have found a simple test for programming aptitude to cleanly separate programming sheep from non-programming goats. In section 5.4 we describe our failed attempt at verifying this result. Section 5.5 is a conclusion on our efforts in this area.

The research presented in this chapter was carried out in collaboration with Bennedsen and is documented in the published papers in chapter 12-14 in the second part of the dissertation. Larsen, an undergraduate student, joined us in the study on mental models described in section 5.4 and chapter 14.

5.1 Related work

Many researchers have investigated programming aptitude aiming at identifying predictors of success for introductory programming courses. The dominating research method is quantitative studies based on statistical regression models [Fox 1997]. An overview of the field is provided by [Bergin et al. 2006]; the essence of Bergin and Reilly's overview, supplemented with other results, is succinctly presented in the following table (Table 5-1).

Researcher(s) and year	***N***	**Language**	**Significant predictors**
[Newsted 1975]	131	FORTRAN	Perceived ability, college GPA[10], and student effort accounted for 41% of the variance.[11]
[Kurtz 1980]	23	FORTRAN	Level of formal reasoning accounted for 63% of the variance.
[Leeper et al. 1982]	92	Not specified	Number of high school English, mathematics, science, and foreign language units, SAT[12] verbal score, SAT mathematics score, and high school rank accounted for 26% of the variance.

[10] College GPA is college grade point average.

[11] 'Factors accounts for *x*% of the variance' means that knowing these factors will lead to predictions that are *x*% more accurate.

[12] SAT is the U.S. College Board's standardized admission test for colleges.

[Barker et al. 1983]	353	Two languages[13]	Piagetian intellectual development accounted for 12% of the variance.
[Konvalina et al. 1983]	382	BASIC	Students who completed the course (228) had significantly more mathematics background than students who withdrew (154).
[Hostetler 1983]	79	FORTRAN	GPA, diagramming and reasoning score, mathematics background, and personality accounted for 77% of the variance.
[Nowaczyk 1983]	286	COBOL	Performance in prior mathematics and English courses accounted for a statistically significant amount of variation (does not say how much).
[Werth 1986]	58	Pascal	Significant correlation found for high school mathematics, hours working at a pert-time job, Piagetian intellectual development, and cognitive style.
[Mayer et al. 1986]	57	BASIC	Word problem translation skills accounted for 50% of the variance.
[Austin 1987]	76	Pascal	High school composite achievement, quantitative and algorithmic reasoning abilities, vocabulary and general information abilities, self-assessed mathematics ability, and measures of an introverted/analytical style and extroverted level accounted for 64% of the variance.
[Cafolla 1988]	23	BASIC	Cognitive development accounts for 34% of the variance.
[Evans et al. 1989]	117	BASIC	High school mathematics courses, prior BASIC experience, hours playing video or computer games accounted for at most 23% of the variance of six different outcome variables (e.g. homework score, mid-term exam, and final exam).
[Gibbs 2000]	50	BASIC	Within a constructivist learning environment, cognitive style was not found to influence programming achievement.
[Goold et al. 2000]	39	C	Dislike for programming, gender, average on other modules, and raw secondary score (English plus other courses) accounted for 43% of the variance.
[Hagan et al. 2000]	97	JAVA	The more programming languages a student knew prior to taking the course, the higher the performance.
[Byrne et al. 2001]	110	BASIC	Mathematics (r = 0.353) and science (r = 0.572) correlates with programming performance.
[Rountree et al. 2002]	472	Java	The strongest single indicator of success was the grade the student expected to get on the course.
[Stein 2002]	160	Java	Students who study calculus do at least as well as students who study discrete math.
[Wilson 2002]	105	C++	Comfort level, mathematics background, and attribution of success/failure to luck accounted for 44% of the variation. The number of hours playing computer games prior to the course had a negative effect while experience of a prior formal programming class had a positive effect.

[13] In [Bergin et al. 2006], Bergin et al. indicate C++ and C as the languages used by Barker et al., but this is due to a misinterpretation of a paragraph on grading; the authors only mention that two languages are used, not which.

[Holden et al. 2003]	159	Java	Prior experience (independent of language) is an advantage in the first course in a programming sequence, but not in later courses.
[Ramalingam et al. 2004]	75	C++	Mental model, self-efficacy, and previous programming and computer experience accounted for 30% of the variance.
[Ventura et al. 2004]	499	Java	Prior programming experience was not a predictor of success for their objects-first CS1.
[Bergin et al. 2005a]	80	Java	Student's perception of his/her own understanding of the course, gender, mathematics score, and comfort level accounted for 79% of the variance.
[Ventura 2005]	499	Java	Percent lab usage, comfort level, and SAT math score accounted for 53% of the variance. Measures of effort are the primary predictors of success followed by comfort level, and then academic predictors (e.g. math) with marginal gains.
[Wiedenbeck 2005]	120	C++	Prior computer and programming experience, self-efficacy, and knowledge organization accounted for 30% of the variance.
[Bergin et al. 2006]	102	Java	Mathematics score, number of hours playing computer games, and programming self-esteem accounted for 80% of the variance. Mathematics score and programming self-esteem were found to have a positive relationship with performance while number of hours playing computer games was found to have a negative effect.

Table 5-1: *Overview of research in programming aptitude*

Even though many of the 25 studies in Table 5-1 have interesting results, it can be hard to know how to apply the results to new educational settings where parameters may differ significantly from the context where the findings were observed. Parameters may be different in many respects:

- course material (e.g. textbook, programming language, development environment)
- course structure (e.g. number of lectures and lab hours)
- course work (e.g. mandatory assignments and project work)
- availability of resources (e.g. support material, support for collaboration, and student/instructor ratio)
- the degree of alignment (concordance between syllabus, course content, and assessment)
- type of assessment (e.g. multiple choice, oral, written, and practical)
- instructor (e.g. teaching experience, familiarity with the subject, and personal attitude)
- students (e.g. age, study seniority, and major)
- external factors (e.g. type of institution, nationality, and culture)

This long list of possible variations among courses indicates how difficult it is to generalise findings from one context to another.

The one finding that seems to be most consistent across various investigations —although not strong— is correlation between mathematics score in high school and performance in CS1, but even this result is questionable. First, we know nothing about the contents and focus of the programming courses where mathematics has been shown to be a predictor of success; traditionally, programming courses practice programming on problems of a highly mathematical nature (e.g. factorial, radix-conversion, exponentiation, and binomial coefficients). In such cases, it may very well be the choice of problems rather than programming per se that causes the result. Second, one might speculate whether the same findings would occur for other high school grades than mathematics; in fact, Rauchas et al. found that "contrary to the generally accepted view that achievement in high school mathematics courses is the best individual predictor of success in undergraduate computer science, success in English at the first-language level in high-school correlates better with actual performance" [Rauchas et al. 2006]. Some of the findings in Table 5-1 show similar results. Third, Ventura's research did not find math ability to be a significant predictor of success in his introductory objects-first programming course: "the current research calls into question the importance of math in the objects-first CS1. First, there was no correlation between the number of math courses a student took in high school and any of the measures of success in the current study. Secondly, SAT math scores always appeared after measures of effort and comfort level. In the overall models the predictive value of SAT math scores was negligible" [Ventura 2005, p. 240].

While local predictors of success have been identified, there is no evidence that these predictors generally hold. To draw local conclusions, we need to perform local studies.

5.2 Local replication of previous studies

With the motivation of improving the course design of our introductory programming course, which is a model-based approach to object-oriented programming with heavy emphasis on the programming process, we decided to study potential indicators of success for the course aiming at answering questions about the relationship between various factors and success in the course. We investigated five factors: mathematical ability, gender, major, student effort, and study seniority, all motivated by previous research in the field [Hagan et al. 2000, Leeper et al. 1982, Ventura 2003, Wilson et al. 2001].

Our research shows that a complete regression model encompassing all five factors accounts for 36% of the variance. However, some factors turned out to be insignificant, and we developed a reduced model with only two factors: math ability and student effort. The reduced model accounts for 24% of the variance. Calculation of the squared partial correlation coefficients reveals that math ability is twice as strong a predictor than student effort: student effort alone accounts for 7% and math ability alone accounts for 15% of the variance. This is similar to the findings of Leeper and Silver who found that math ability accounted for 14% [Leeper et al. 1982]. Gender, major, and study seniority were not significant, neither at the 95% confidence interval, nor at the 90% interval. Details of this research are documented in the paper in chapter 12 and published in the proceedings of ICER 2005 [Bennedsen et al. 2005b].

As many others, we found math ability to account for a fair share of the variance in programming performance. Realizing that it hardly is specific math competencies such as calculus, geometry, or trigonometry that have a positive impact on programming performance, we hypothesised that it is the more general notions of abstraction and reasoning ability that indicates programming aptitude. Adherence to strict notations with a formal syntax and semantics may also be something that comes more natural to people with good mathematical abilities, but we have not pursued that line of thought.

5.3 Abstraction ability

Many computer science educators argue that abstraction is a core competence [Alphonce et al. 2002, Kurtz 1980, Nguyen et al. 2001, Or-Bach et al. 2004, Sprague et al. 2002]. Nguyen and Wong [Nguyen et al. 2001] claim that it is difficult for many students to learn abstract thinking; at the same time they claim abstract thinking to be a crucial component for learning computer science in general and programming in particular. The authors describe an objects-first-with-design-patterns approach to CS1 with a strong focus on abstract thinking and development of the students' abstractive skills. In [Or-Bach et al. 2004] the authors argue that abstraction is a fundamental concept in programming in general and in object-oriented programming in particular. Clearly, abstraction and abstract thinking are fundamental concepts in computer science and key components of learning programming. For programming education (and CS education in general) it is therefore mandatory to explicitly aim at the development of the students' abstractive skills. But furthermore, we anticipate general abstractive skills —abstraction ability— to be an indicator of success for learning programming. Our hypothesis is therefore: *General abstraction ability has a positive impact on programming ability*.

5.3.1 Operationalization of hypothesis

To operationalize the first part of our hypothesis we need to define what we mean by 'abstraction ability' and how it can be measured. [Or-Bach et al. 2004] define abstraction ability in terms of object-oriented programming. However, abstraction ability is a much more general skill often defined as part of the cognitive development stage of a person as described by Inhelder and Piaget in [Inhelder et al. 1958]. Our approximation of abstraction ability is based on Adey and Shayer's theory of cognitive development [Adey et al. 1994, Shayer et al. 1981]; this theory is a refinement of Inhelder and Piaget's stage theory. Adey and Shayer define eight stages of cognitive development (Table 5-2).

Identification	**Description**
1	Pre-operational
2A	Early concrete
2A/2B	Mid concrete
2B	Late concrete
2B*	Concrete generalisation
3A	Early formal
3A/3B	Mature formal
3B	Formal generalisation

Table 5-2: *Stages of cognitive development*

We use Adey and Shayer's stage model of cognitive development to characterize the students' abstraction ability. To measure abstraction ability defined in this way, we use a reasoning ability test developed by Piaget, and refined by Adey and Shayer, for testing at the higher end of the stage model. In this research, we use the results from the final exam of the introductory programming course as an indicator of the students' programming ability. Thus, the specific research question we have investigated is: *Is there a positive correlation between the stage of cognitive development and the student's result in the final exam of a model-based introductory programming course?*

5.3.2 Students and data

The investigation was carried out in the fall of 2005; 256 students took the final exam, and 145 of these had volunteered to participate in the research. The distribution of the subjects' cognitive development according to the test and the distribution of the grades of the final exam are shown in Figure 5-1 (in both diagrams, the unit of the y-axis is (the number of) subjects).

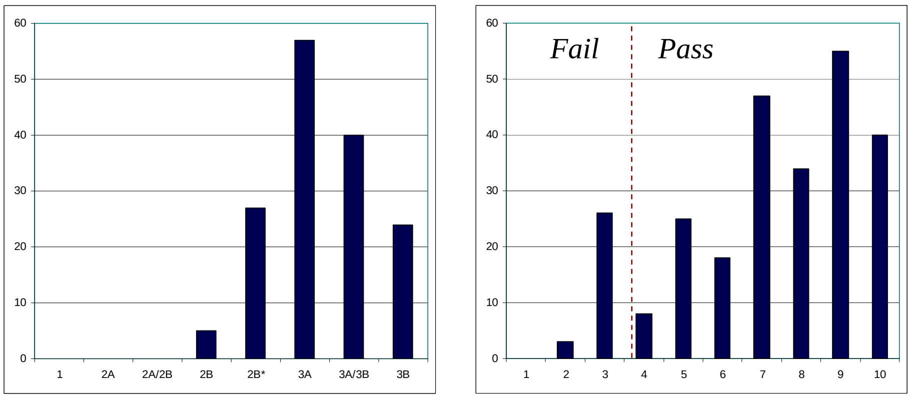


Figure 5-1: *Distribution of subjects with respect to stage of cognitive development and grade in the final exam*

We were concerned that the cognitive development would not follow a normal distribution since the test is developed for pupils in the age range of 5 to 16, and we are using it for students in the range 18 to 24. Shayer and Adey found that 30% of the pupils were at stage 3A at the age of 16 and 10% at stage 3B. However, they also found that the curve describing the progression of stages was very flat at that stage [Adey et al. 1994 p. 40]. We found 37% of the students to be at stage 3A and 16% to be at stage 3B. Overall we found our observations of cognitive development to follow a normal distribution as indicated by the left histogram of Figure 5-1.

5.3.3 Findings

We found no correlation between the stage of cognitive development and the students' results at the final exam. This was a major surprise, particularly because others have reported that cognitive development accounts for 34% of the variance of the exam score [Cafolla 1988]. Cafolla's study was based upon students learning programming in BASIC; it seems unlikely that programming in BASIC should require a higher degree of cognitive development than object-oriented programming.

We have checked for correlation between cognitive development stage and performance in other subjects than introductory programming. We have checked math grade from high school and performance in three courses following the introductory programming course (the second programming course, a course on computer architecture, and a course on regularity and automata theory); in all four cases the result was the same: no correlation between cognitive development stage and performance in the course!

5.3.4 Discussion

The result of this study is most surprising. From the outset we were certain that students at a higher stage of cognitive development would get higher scores in the final exam of the introductory programming course. It is not so.

There can be several explanations to this. In our programming course, coding is prioritized over design. The cognitive requirements are therefore relatively low, and apparently there are other factors that influence the students' success. Another potential explanation is the instrument used to assess the cognitive stage: the pendulum test. The pendulum test measures the student's ability to control independent variables in a reasoning task. It could be that this particular competence is not prominent in our course. Finally, of course, it is questionable to which extend the result of the final exam is a reasonable measure of a student's ability to learn programming.

Further details of this research are documented in the paper in chapter 13 and published in the SIGCSE Bulletin [Bennedsen et al. 2006a].

5.4 Mental models

In a teaser email circulated in late 2005, shortly before the PPIG workshop in January 2006, Bornat wrote: "We have a scientific breakthrough that we'd like to announce at your little PPIG. The breakthrough is that Saeed has a test which picks out, with 100% accuracy, those people who have a chance of learning to program and rejects, with 100% accuracy, those who have no chance. Don't believe it? Neither did I, at first, but it's true. And I'm not telling you, before the little PPIG, just how it's done. But of course I will tell you all there." We learned about the test in conjunction with the PPIG workshop in January 2006. Having searched for predictors of success for introductory programming courses, we were certainly intrigued by the promotion material, and we decided to try to verify Dehnadi and Bornat's findings.

5.4.1 The test instrument

Dehnadi and Bornat [Arthur 2006, Dehnadi 2006b, Dehnadi et al. 2006a] claim they have found a way to identify students who will not succeed in learning programming. Based on a test of 60 students, they claim: "We have found a test for programming aptitude, of which we give details. Remarkably, we can predict success or failure even before students have had any contact with any programming language, and with total accuracy" [Dehnadi et al. 2006a].

Dehnadi and Bornat classified students according to their consistency in answering a set of similar questions. The overall hypothesis is that consistent students and consistent students only will be able to learn to program. To determine consistency, Dehnadi and Bornat used a questionnaire with 12 small Java programs. Each program consists of two variable declarations and one, two, or three assignment statements; Figure 1-1Figure 5-2 shows a sample.

5. Read the following statements and tick the box next to the correct answer in the next column.	The new values of a and b are:	
	☐ `a = 30`	`b = 50`
	☐ `a = 10`	`b = 10`
	☐ `a = 20`	`b = 20`
`int  a = 10;`	☐ `a = 10`	`b =  0`
`int  b = 20;`	☐ `a =  0`	`b = 20`
	☐ `a = 30`	`b =  0`
`b = a;`	☐ `a = 40`	`b = 30`
`a = b;`	☐ `a =  0`	`b = 30`
	☐ `a = 20`	`b = 10`
	☐ `a = 30`	`b = 30`
	☐ `a = 10`	`b = 20`
	Any other values for a and b:	
	`a =`	`b =`
	`a =`	`b =`
	`a =`	`b =`

Figure 5-2: *A sample question from Dehnadi and Bornat's questionnaire*

Dehnadi and Bornat have identified 11 different mental models of assignment which are captured by the 11 options in the questionnaire (along with the last option: *other*); each option corresponds to a certain mental model of assignment. The questionnaire contains 12 questions similar to the one in Figure 5-2, giving rise to a 12-tuple describing the mental models applied by a student (e.g. (m_7, m_3, ..., m_7)) where m_i represents a mental model. The 12-tuple is used to *assign* each student to one of three categories:

- The *consistent* group. The students who use the same mental model for most of the questions (irregardless of which model).
- The *inconsistent* group. The students who use varying mental models for the questions.
- The *blank* group. The students who refuse to answer the questions.

In [Dehnadi 2006b], the authors write: "The consistent/inconsistent/blank assignment which is the basis of our preliminary result was rather subjective" and the authors devise a more objective instrument for categorisation of the students—an instrument which we shall use in our investigation. Dehnadi and Bornat found that 44% of their students belong to the consistent group, and 39% belong to the inconsistent group; 8% left the questionnaire blank (the remaining 9% are missing).

Dehnadi and Bornat conclude that the test, although not perfect, is the first test to be able to claim any degree of success. It is indeed very interesting *if* Dehnadi and Bornat have found a predictive test as they describe.

5.4.2 Failure of verification

We have conducted a study to verify Dehnadi and Bornat's findings by examining the predictive power of a student's mental model for his or her success in learning introductory programming. The hypothesis is that there is a positive correlation between a student's mental model and the student's ability to learn programming. The specific research question we investigated is the following: *Is there a correlation between the students' consistency in the mental model applied in questionnaire and their performance in the final exam of a seven-week introductory, model-based, object-oriented programming course?*

The population for this study was 142 students following the introductory programming course in the fall of 2006; of the 150 students who volunteered to participate at the beginning of the course, 142 attended the final exam.

The students answered the questionnaire in the first week, before the assignment statement was taught. To determine the consistency of the mental model for each of the students, we used the categorization instrument proposed by Dehnadi [Dehnadi 2006b]. From the 12-tuple that describes the mental models applied by a student in the questionnaire, we divided the students into five categories C_i, $0 \leq i < 5$, of decreasing consistency, C_0 being the most consistent category and C_4 the least consistent category. A student is in consistency category C_0 if at least eight mental models in the student's 12-tuple are identical. For the coarse-grained categorization used by Dehnadi and Bornat, students in C_0 are considered consistent while students in any of the other categories are considered inconsistent. For further details, see [Dehnadi 2006b].

The final exam resembles an ordinary lab session. The students are tested in groups of up to 25 at a time. The effective examination time is 30 minutes (occasionally, for various reasons, we allowed a bit more time). There are *two checkpoints* in the assignment which consists of ten subtasks: one after task three and one after task eight. The students are instructed to call upon an examiner to demonstrate their solutions when they reach either of the checkpoints. For each student, we noted the elapsed time at both checkpoints as well as when (if) they finished the assignment (*first interval*, *second interval*, and *final time*), thus providing a rough measure of the student's efficiency and competence.

The grading schema for the exam is a binary pass/fail which is too coarse-grained to allow for statistical analysis. Therefore, we subdivided the students into four groups, G_i, $0 \leq i < 4$. G_0 represents the students that failed the exam; G_1 represents the students who barely passed the exam (i.e. reached the second checkpoint in the very last minute), G_2 represents the students who produced an average performance (i.e. reached the second checkpoint in due time but did not finish the assignment), and G_3 represents the students who finished the assignment within the time limit with a program that fulfils the complete specification.

The distribution between consistent and inconsistent broken down to the exam result and prior programming experience is shown in Table 5-3.

N = 142	**Consistent**	**Inconsistent**
Total	124	18
Pass at the final exam	120	16
Fail at the final exam	4	2
Prior programming experience	85	2

No prior programming experience	39	16

Table 5-3: *Number of consistent and inconsistent students*

In order to verify Dehnadi and Bornat's findings, we have used a Pearson correlation coefficient test to find if, for students with no prior programming experience, there is a significant correlation between the consistency level and the grading level (according to the *C*- and *G*-categories described above. The correlation coefficient is –0.072 and we conclude that there is no correlation between consistency of the mental model and performance in our introductory programming course, i.e. we cannot verify Dehnadi and Bornat's findings.

To take a closer look at this contradictory result, we have tested for correlation for a more fine-grained partitioning than the five competence-levels and four grading levels applied above. We made a more fine-grained partitioning of the mental models by refining the C_i categories: C_i represents the students' whose maximum number of answers of the same mental model equals *i*, thus providing 13 different categories of mental models. Similarly, we have refined the G_i categories to reflect the students' performance according to the second interval (the time elapsed when reaching the second checkpoint), i.e. G_i is the students for whom the second interval is *i* minutes. The distribution of the data certainly does not indicate a correlation (see Figure 5-3). A Pearson correlation test confirms this impression with the same result as before (P=–0.075).

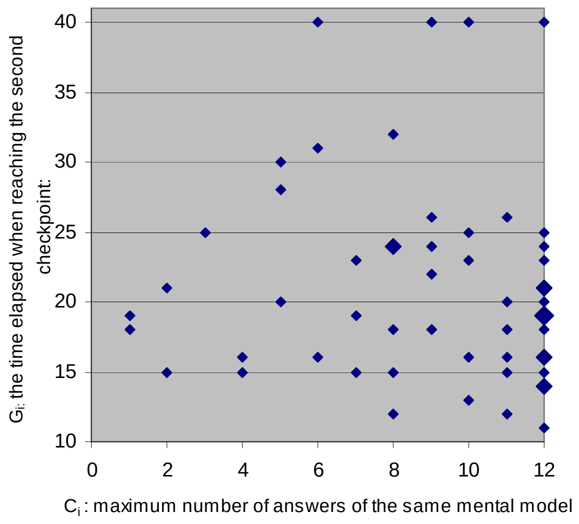


Figure 5-3: *Second interval versus maximum number of identical mental models in 12-tuple*

Our result is a clear and unequivocal rejection of the research question: there is absolutely no correlation between students' consistency of the mental model applied in the questionnaire and their performance in the final exam of a seven-week introductory, model-based, object-oriented programming course. If the hypothesis of positive correlation between a student's mental model and ability to learn programming is to be confirmed, it requires an interpretation of the mental model which is different than the one reflected in Dehnadi's questionnaire or another interpretation of ability to learn programming than the one reflected by the exam of our introductory programming course.

Our unequivocal result gives rise to a number of questions. One question is whether Dehnadi and Bornat's interpretation of their results is viable. Another question is the validity of the test instrument and speculations about other and better test instruments. These questions and further details of this research are addressed in the paper in chapter 14 which is submitted for ITiCSE 2007.

5.5 Conclusion

In his regular column in SIGCSE Bulletin [Lister 2005a], Lister addressed the issue of programming aptitude. Lister phrases the challenge as identifying a "programmer quotient" (PQ). Inspired by the concept of an IQ, a PQ would be "a measure of programming ability, relative to the whole population, that would remain the same independent of programming experience". However, the long history of attempts to find predictors of success in programming has so far not managed to come up with anything like this. Although recent linear regression models have accounted for almost half the variation of student performance, such models are only a weak indicator of an individual's PQ. Lister concludes: "One thing is certain: there is no 'silver bullet'. If there was a silver bullet, then we would already have found it" [Lister 2005a].

Our own research in this area has neither revealed new predictors of success nor been able to confirm the findings of others except for a weak impact from math grade in high school, but this impact might exist for other grades from high school as well as indicated by the findings of [Rauchas et al. 2006] that English at the first language level in high school correlates better with performance in programming than does math ability.

Our results may be regarded as negative results, but we don't consider it that way. In the light of our thesis that everybody can learn to program, the results are quite encouraging, and we speculate whether our special flavour of an introductory programming course — a model-based approach to object-oriented programming with heavy emphasis on the programming process — has something to do with this. However, further research is required to conclude anything along these lines.

In closing, we would like to chime in with Lister [Lister 2005a] and Hazzan et al. [Hazzan et al. 2006] and conclude that perhaps the way forward is not to look for more statistical variables to add to regression models, but instead to conduct qualitative research based on observations of and interviews with students. Eventually, qualitative research might provide new insights that lead to factors for incorporation into predictive models such as a PQ.

6 Programming Methodology

It is time to address the fourth research question:

> **Research question 4** (Q_4): How does best-practice in modern software development relate to the research area of programming methodology? The question is refined to four more specific questions:
>
> $Q_{4.1}$: How has programming methodology influenced programming education in the past?
>
> $Q_{4.2}$: How can we characterize best-practice of modern software development?
>
> $Q_{4.3}$: How does best-practice in modern software development relate to programming methodology?
>
> $Q_{4.4}$: Can we provide a characterization of the programming process that unifies programming methodology and best-practice of modern software development?

In this chapter, we address $Q_{4.1}$ and $Q_{4.2}$ by presenting a historical perspective on the role of programming methodology in programming education and providing a brief overview of best-practice of modern software development. The two remaining questions are addressed in chapter 7.

Under the headline, *Where is Programming Methodology These Days?*, David Gries wrote an invited editorial in the December 2002 issue of *inroads* – SIGCSE Bulletin [Gries 2002]; the occasion was Edsger W. Dijkstra's passing on 6 August the same year. Gries wrote:

> *For several years, I have been thinking about the influence —or lack thereof— that the field of programming methodology has had on programming and how we teach it [...].*
>
> *In terms of knowledge —facts, definitions, particular algorithms, etc. — there has been much progress. The development of OO has helped tremendously, in the sense that it has given us a tool for organizing our programs and our thoughts about them [...].*
>
> *However, programming is more than a bunch of facts. It is a skill, and teaching such a skill is much harder than teaching physics, calculus, or chemistry. People expect a student coming out of a programming course to be able to program any problem. No such expectations exist for a calculus or chemistry student. Perhaps our expectations are too high. Compare programming to writing. In high school, one learns about writing in several courses. In addition, every college freshman takes a writing course. Yet, after all these courses, faculty member still complain that students cannot organize their thoughts and write well! In many ways, programming is harder than writing, so why should one programming course produce students who can organize their programming thoughts and program well.*
>
> *In any case, teaching programming as a skill means more than teaching facts. It means teaching students how to think when designing, developing, testing, and debugging a program. It means extending their*

problem-solving abilities. It means giving them effective strategies and principles that will shorten programming time and reduce the need for debugging (but not for testing). It means teaching good thought habits. In addition, it means teaching basic theory that provides understanding and that they can put into practice.

In this regard, based on the textbooks I have looked at, we are failing. Few texts teach important basic theory. Few texts discuss important strategies and principles, and if they do, they don't practice them within the text; instead, they are caught up with teaching more and more features. Few texts discuss program development; instead, they simply present programs and leave development principles and strategies for students to discover on their own [...].

Throughout his career, Dijkstra thought deeply about programming methodology [...], perhaps more than anyone else. This is the time to pause and reflect on whether we are making use of all that he has given us.

So, let us pause and reflect on where programming methodology is in education these days, where it was in the past, and where it ought to be.

6.1 A contemporary perspective

To a large extent, the joint ACM and IEEE curriculum recommendations set the scene for textbook contents and teaching practice. The Computing Curricula 2001 (CC2001) [ACM 2001] characterize the current state of affairs of programming courses as follows:

Programming courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying algorithmic skills. This focus on details means that many students fail to comprehend the essential algorithmic model that transcends particular programming languages.

Moreover, 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. Such courses thus risk leaving students who are at the very beginning of their academic careers to flounder on their own with respect to the complex activity of programming.

Apart from echoing the observations of Gries, the excerpt from CC2001 exhibits the traditional interpretation of "skills" as algorithmic problem solving. However, as Gries indicates, teaching programming as a skill is much more than that.

In the editorial, Gries demonstrates what he means about teaching principles and strategies used during program development by presenting a derivation of selection sort (instead of just presenting a *fait acompli*) using informally the theory of loop invariants and the checklist for developing or presenting a loop. The point is: to learn an algorithm, memorize *concepts*, not code. The basic techniques are stepwise refinement and separation of concerns. The theory that Gries applies was developed by Dijkstra in the mid-1970s [Dijkstra 1975, Dijkstra 1976] and polished for novices more than a quarter of a century ago [Gries 1981]; however, it is still not part of the common interpretation of the vague terms used in curriculum descriptions (e.g. *algorithmic problem solving* and *structured decomposition*).

To provide a more complete picture, we prefer to characterize teaching programming as a skill as:

- *describing* strategies, principles, patterns, and techniques of program development
- *demonstrating* how to apply these in action, and
- giving novices the opportunity to *practice* while receiving feedback in order to improve their skills.

The strategies, principles, patterns, and techniques of (object-oriented) program development encompass:

- *development strategies* (e.g. incremental development, responsibility-driven development, model-driven development, test-driven development),
- *design and programming principles* (e.g. the single-responsibility principle, the information hiding principle, the substitution principle, the mañana principle),
- *problem-solving patterns* (e.g. design patterns, algorithmic patterns, elementary patterns, variable patterns),
- *design and programming techniques* (e.g. CRC-cards, design-by-contract, system invariants, class invariants, loop invariants, factoring out of inner loops, factoring out of heavy functionality).

Complementary to these are skills related to finding and correcting errors, refactoring program design, and optimizing the performance of programs.

Clearly, we cannot and should not teach all of this at once, but we need to understand the nature of modern program development in order to devise *an instructional design for programming education* that can bring students from the level of novice toward the level of expert. If we also want the instructional design to be *efficient* (i.e. advance the programming skills to the required level for as many students as possible), we must take modern cognitive learning theories into account.

6.1.1 CC2001 on teaching programming skills

To get closer to an answer to the question, *Where is Programming Methodology These Days?*, let us see what CC2001 says about programming skills. The programming fundamentals (PF) part of the CS body of knowledge has the following to say about programming skills:

- PF1: *Topic*: Structured decomposition. *Learning objectives*: Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
- PF2: *Topic*: Problem-solving strategies; the role of algorithms in the problem-solving process; debugging strategies. *Learning objectives*: Discuss the importance of algorithms in the problem solving process; create algorithms for solving simple problems; describe strategies that are useful in debugging.
- PF4: *Topic*: Divide-and-conquer strategies; recursive backtracking. *Learning objectives*: Describe the divide-and-conquer approach; discuss problems for which backtracking is an appropriate solution.

We shall not discuss the formulation of the learning objectives; for our purpose it suffices to note that there is very little focus on programming skills; the CC2001 recommendations and our description of programming skills above are miles apart. Except for a bit about recursion and debugging, all that is mentioned is the vague terms of structured decomposition and problem solving strategies—abstract terms that, without a modern interpretation, easily become echoes from a distant past or even worse: buzzwords.[14]

Of course they need not become buzzwords in the classroom; it could be that textbook authors and educators provide the contemporary interpretation necessary to revitalise the CC2001 recommendations on programming skills. However, as the following two subsections indicate, this does not seem to be the case.

6.1.2 A textbook survey

In 2002, Pauline H. Mosley conducted a survey of the most popular first-level computer science texts on object-oriented programming adopted by universities [Mosley 2002]. Mosley found that the average number of pages used for elaboration on design (the term that comes closest to programming methods and programming skill) is only two pages, and on design procedures less than one page! Mosley concludes: “This lack of elaboration may be the reason why practitioners stated that they were unclear as to how to apply design concepts to various software engineering problems.”

The Unified Modeling Language is mentioned in only two of the ten books (two pages in one book and 29 pages in another). Mosley speculates: “The absence of the Unified Modeling Language is perhaps an indicator of the complexity of design, or that learning about system design is something different from learning how to program. It could be that instructors perceive design as a separate subject, unrelated to object technology.”

Mosley concludes the survey: “Academicians feel that design should be taught in conjunction with object-oriented programming, at least in the best of all possible worlds. This suggests the question that if programming textbooks are not addressing design, perhaps college instructors are incorporating design concepts on an ad hoc basis. Perhaps students learning Java in college are exposed to more design along the way” [Mosley 2002].

Mosley’s findings concur with Kölling’s observations: In [Kölling 2003b], a survey of 39 major selling textbooks on introductory programming was presented. The overall conclusion of the survey was that all books are structured according to the language constructs of the programming language; the process of program development is often merely implied rather than explicitly addressed.

While textbooks do not seem to address programming as a skill, it might be the case, a Mosley suggests, that educators incorporate design concepts on an ad hoc basis.

[14] According to Wikipedia, buzzwords are words or terms that, although apparently ubiquitous in a certain environment, often have unclear meanings.

6.1.3 Two educator surveys

In 2004, Nell Dale conducted a survey among 350 CS educators from the SIGCSE community [Dale 2004]. In [Dale 2005], Dale reported that on the question “Should problem solving and design be explicitly taught?”, 78% responded that it is very important to teach these topics explicitly; while 16.7% responded that it is important to teach problem solving and design, but that a few concrete examples should suffice. Dale concludes: “Algorithm development, regardless of the design methodology used, is clearly a focus for most instructors in CS1: 80.3% described it as a thread spread throughout multiple lectures and 13.3% reported devoting several lectures.” In [Dale 2006], Dale reported from the same survey on the topic that educators felt was the most difficult for beginning students. Four categories emerged, the first being problem solving and design. In summary: problem solving and design is important to teach, but it is very difficult for beginning students. However, more interesting is that neither the survey designer nor the respondents are explicit about what they mean by problem solving and design.

In [Raadt et al. 2004a], the authors report on the findings of a census of introductory programming courses; 85 courses from Australian and New Zealand universities are included. Instruction on problem-solving strategies varies greatly in the courses covered by the census, as does estimates of the proportion of lecture time devoted to the instruction of problem-solving strategies. Some participants indicated that teaching problem-solving strategies was not a part of their course; several of these instructors felt that the problems used in their teaching were not of a large enough scale to warrant teaching problem-solving strategies explicitly. Others said that their entire lecture time focused on teaching of problem-solving strategies; these instructors did not distinguish explicit teaching of problem-solving strategies from other parts of their teaching. The authors conclude that the variation may be due to instructors not having a common definition of what is involved in the explicit teaching of problem-solving strategies.

From this and the previous section, we conclude that to the extent that programming as a skill is addressed in textbooks and by educators in their teaching, it seems to be based upon vague interpretations of such terms as problem solving and algorithm development.

6.1.4 The programming education research perspective

To complete the picture, we will take a look at how programming education research addresses the issue of teaching programming as a skill.

A minor but remarkable collection of programming education research from the past ten to fifteen years concerns *a pattern-based approach to instruction* which utilize a shift from emphasis on learning the syntactic details of a specific programming language to the development of general problem-solving and program-design skills [East et al. 1996] (see also the parts on *case studies and apprenticeship* and *patterns* (*for schema creation*) in section 4.2.3). The approach was motivated by a shared perception that too many students cannot write reasonable programs even after one or two semesters of programming education. The approach was also motivated by the fact that “textbooks address top-down design by admonishing students to break larger problems into smaller problems and by giving static examples that illustrate a very dynamic process." A static program example presented in a textbook reveals nothing about the process of developing the program. Consequently, students get no insight into how problems can be broken down and solved. The last motivating factor was an urge to take pedagogical issues into account: "There is indeed little discussion of the teaching of programming that relates to pedagogy and almost none that address how the process of learning might or should affect instruction." [East et al. 1996].

In the light of section 3.1, we may characterize the emerging pattern-based approach to instruction as a way of compensating for novices' lack of schemas by identifying domain patterns and make them first-class citizens in education—by injecting schemas directly into memory, so to speak. And from section 3.3, we may well expect a pattern-based approach to instruction to be superior to the traditional syntax-oriented approach; experience as well as solid research (section 4.2.1) tells us that students don't build suitable schemas (or identify patterns) by themselves no matter how many programs they develop. But, as in the case of the alternating rule in the number transformation problem of section 3.2, the best solution is there for the picking: tell them! Make the patterns explicit to the students so they can focus their resources and optimize learning.

Others have practiced a pattern-based approach to programming education, and much earlier than the references mentioned earlier. At DAMI[15], Schmidt introduced algorithmic patterns in CS1 [Schmidt 1980]. An algorithmic pattern is an abstract algorithm with variation points; the abstract algorithm captures the behaviour of a family of algorithms. Concrete algorithms are achieved by providing concrete bindings to the variation points in the abstract algorithm. An example is an abstract algorithm for searching which can be concretized to various concrete searching algorithms, e.g. linear search, binary search, and randomized search. To defuse the formalism of invariant techniques, and thereby making them more accessible and applicable to novices, Schmidt [Schmidt 1980] introduced pictures as invariants. Schmidt never published this, but Astrachan did a decade later [Astrachan 1991].

Inspired by Polya's classic book on problem solving in mathematics [Polya 1957], Dromey wrote a wonderful textbook, *How to Solve it by Computer* [Dromey 1982]. In the preface to the book, the author writes: "many beginners in computer science stumble not because they have difficulty with learning a programming language but rather because they are ill prepared to handle the problem-solving aspects of the discipline. [...] If we can develop problem-solving skills and couple them with top-down design principles, we are well on the way to becoming competent at algorithm design and program implementation. Emphasis in the book has been placed on presenting strategies that we might employ to '*discover*' efficient, well structured computer algorithms" [Dromey 1982 p. xiii]. Unfortunately, Dromey's book never became widely used.

The importance of teaching underlying patterns and principles was stressed by Knudsen and Madsen in the context of teaching object-oriented programming languages. In [Knudsen et al. 1988], the authors argue that it is

---

[15] DAIMI is the pet name for the Department of Computer Science at the University of Aarhus.

vital to provide a theoretical foundation —a conceptual framework— upon which the students can base their understanding of concrete languages and language constructs: “The prime message to be told is that working from a theoretical foundation pays off. Without a theoretical foundation, the discussions are often centered around features of different languages. With a foundation, discussions may be conducted on solid ground. Furthermore, students have significantly fewer difficulties in grasping the concrete programming languages when they have been presented with the theoretical foundation than without it” [Knudsen et al. 1988].

A recent comprehensive approach to pattern-based instruction is described by Muller [Muller 2005b, Muller et al. 2004, Muller et al. 2005a]. Muller’s approach utilizes a course orientation that shifts from emphasis on learning the syntactic details of a specific programming language to the development of general problem-solving and program-design skills. Apart from describing the pattern-based approach, Muller also characterizes the current state of affairs in introductory programming courses: “courses often focus more on the programming language syntax rather than on developing algorithmic problem-solving skills. The main organizational guideline for preparing syllabus and textbooks usually focuses mainly on the programming language features. Algorithmic problem solving is commonly practiced through exercises in the use of language features that have been taught beforehand. Most of the attention and effort are devoted to language details and to running programs. General problem-solving issues are discussed but often not in the explicit and structured manner needed” [Muller 2005b]. A consistent observation is made by Raadt Toleman, and Watson: “Traditionally [programming] strategies are taught mainly through implicit instruction where novices undertake extensive sets of practical exercises. Students are expected to learn from their successes and failures in order to develop highly structured problem-solving strategies, and so become able to solve future problems. Using this implicit approach to problem-solving instruction can mean some novices do not develop appropriate skills and risk failing courses or being unprepared for later programming practice” [Raadt et al. 2004b]. Raadt et al. summarize: “It is our belief that the current implicit approach to instruction could be replaced with a carefully structured methodology which incorporates explicit instruction of problem-solving strategies”.

From the above, we conclude that except for a few far-sighted researchers that promote various forms of patterns and techniques, programming skills and the programming process does not have a solid seat in the joint awareness of what to teach in introductory programming courses. In general, there seems to be a restricted, vague, and highly varying interpretation of what constitutes (teaching of) problem-solving strategies. In the next section, we explore the background for this state of affairs.

6.2 A historical perspective

Programming methodology emerged in the late 1960s through accomplishments of prominent people like Dahl, Dijkstra, Hoare, Naur, and Wirth who all later became Turing award winners [Dijkstra 1969, Naur 1966, Naur 1972, Dahl et al. 1972, Wirth 1971, Wirth 1974]. The programming methodology fashion of the time became known as *structured programming*, *stepwise refinement*, and *top-down programming*. IFIP established working group 2.3 on programming methodology in 1969 [IFIP 2006]. A comprehensive collection of seminal publications by members of WG 2.3 is provided in [Gries 1978].

6.2.1 Emphasis in education

In 1974, Knuth wrote: “A revolution is taking place in the way we write programs and teach programming, because we are beginning to understand the associated mental processes more deeply,” [Knuth 1974]. Indeed, the development in programming methodology and the notions of structured programming, stepwise refinement, and top-down design spawned a lot of interest and activities in programming education and programming education research during the 1970s and 1980s [Bagert 1988, Basili et al. 1974, Danielson et al. 1975, Dupras et al. 1984, Riley 1981, Taylor 1977, Weiner 1978]. Ulloa provide a good overview of initiatives from the first of the two decades [Ulloa 1980]. The growing awareness of the importance of the introductory programming course is reflected by Dupras et al.: “The first contact of the student with programming should be planned very carefully. This must be reflected in the choice of instructor, the choice of textbook, the choice of methodology, and the choice of programming language, etc. The first programming language course must concentrate on teaching a programming methodology rather than a programming language. The concern for program correctness must be taught from the very first course of programming. This means that the difference between early programming courses and advanced programming courses should not be in how much emphasis is placed on program correctness but rather in the complexity of the applications to be programmed” [Dupras et al. 1984].

In 1988, Means conducted a textbook analysis on ten introduction-to-programming textbooks to determine the development of content over time. The books were divided into two categories: the first category consisted of three pairs of books with first editions from 1983 or earlier and second editions from 1985 or later; the second category consisted of four books of different authors, two from 1978 and two from 1986. “An increased emphasis on the problem solving aspect of computer science was noted in the increase in space devoted to this topic [...]. Algorithmic development also demonstrated an increase [...]. Top-down design also increased in space [..]. The changes may reflect a change in the view that a competent programmer needs to know more than just the syntax of the language” [Means 1988].

Clearly, there was a strong and increasing emphasis on problem solving, divide-and-conquer, top-down design, and stepwise refinement in introductory programming education during the 1970s and 1980s and this emphasis was echoed in programming education research of that time, e.g. [Rist 1989, Soloway et al. 1983, Soloway et al. 1989, Spohrer et al. 1986] (for further references, see section 4.2.1).

While many addressed stepwise refinement, top-down design, and divide-and-conquer in programming courses, textbooks, and programming education research, there was also a growing awareness that students had a hard time grasping and applying the principles of strict top-down development and that teaching programming as stepwise refinement and top-down design perhaps was fundamentally flawed.

As early as 1980, Ulloa identified the difficulty of learning top-down programming as one of three major problems of teaching problem solving to novices [Ulloa 1980]. In 1981, Riley wrote: "it is evident that many students entering college have problem-solving skills that are woefully inadequate" [Riley 1981]. In 1989, Gantenbein observed: "Attempts have been made to teach problem solving in introductory programming texts. Unfortunately, most of the time the rules are too general to be fully understood or too specific to be widely applicable" [Gantenbein 1989]. In 1986, Hoare —in his usual succinct style— delivered the ultimate verdict:

You cannot teach beginners top-down programming, because they don't know which end is up.

— C.A.R. Hoare, 1986[16]

In 1993, Pattis followed up on Hoare's verdict: "an emphasis on early design is misplaced. It is too much to ask beginning students to learn good design principles first", and Pattis concluded: "it is not a good idea for them to apply stepwise refinement solely as top-down design" [Pattis 1993]. Pattis did more than point out the problem, he also devised a solution. We return to the latter part of Pattis' contribution in section 6.2.3.

The good thing about the development of programming education in the 1970s and 1980s was that early achievements of programming methodology were adopted by the community and quickly found their way into the curriculum of the introductory programming course. The unfortunate thing was that the adopted approach was a narrow and flawed interpretation of structured programming and stepwise refinement which —for good reasons— did not succeed.

6.2.2 Two misconceptions

In 1975, Denning pointed out two serious and prevalent misconceptions about structured programming that he had observed turning up with "alarming frequency" in writings on the topic [Denning 1975]. First, structured programming is not necessarily top-down programming, but rather, it encompasses any approach leading to well-structured modularization. Second, he emphasized that it is not true that the means to achieve a structured program have always to be structured themselves. Denning wrote: "It is a misconception that top-down programming is the only way in which a good program can be developed and that structured programming limits us to this scheme of modularization". Denning recognizes that oft-quoted texts and writings contain many examples of programs whose modularization is explained by this technique, e.g. [Dijkstra 1969, Wirth 1971, Wirth 1974]. Denning points out: "The confusion is in failing to distinguish the product from the process that created it".

However, from the initial writings of Dijkstra and Wirth, one can easily be led astray to conclude that top-down stepwise refinement *is* the way to compose programs and teach programming. Before the first example of stepwise program composition in his *Notes on Structured Programming*, Dijkstra

[16] Tony Hoare's verdict was delivered at a 1-1 meeting with Richard Pattis at University of Washington on 23rd October 1986.

writes: “Instead of presenting (as a ready-made product) what I would call a well-structured program I am going to describe in very great detail the composition process of such a program. I do this because programs are not there: on the contrary they have to be made” [Dahl et al. 1972 p. 26].

In his seminal paper, *Program Development by Stepwise Refinement*, Wirth characterized stepwise refinement as follows [Wirth 1971 pp. 227-228]:

1. Program construction consists of a sequence of refinement steps. In each step a given task is broken up into a number of subtasks. [...]
2. The degree of modularity obtained in this way will determine the ease or difficulty with which a program can be adapted to changes or extensions [...]
3. During the process of stepwise refinement, a notation that is natural to the problem in hand should be used as long as possible. The direction in which the notation develops during the process of refinement is determined by the language in which the program must ultimately be specified, i.e. with which the notation ultimately becomes identical. [...]
4. Each refinement implies a number of design decisions based upon a set of design criteria. [...] Students must be taught to be conscious of the involved decisions [...]. In particular, they must be taught to revoke earlier decisions, and to back up, if necessary, even to the top.

Three years later, Wirth recognized (as a post rationalization?) that those who read his examples of programming by stepwise refinement might conclude that the process as well as the product had a top-down structure; as a consequence Wirth wrote: “I should like to stress that we should not be led to infer that actual program conception proceeds in such a well organized, straightforward, ‘top-down’ manner. [...] But this neat, *nested factorization* of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness” [Wirth 1974 p. 251]. Denning concluded: “By all means, let us advise our fellow programmers to evolve programs explainable by a top-down method. But let us not confuse the end with the means” [Denning 1975].

History has revealed that Wirth’s “disclaimer” and Denning’s advice were both ignored by the programming education community (section 6.2.1).

6.2.3 Stepwise enhancement

As mentioned at the closing of section 6.2.1, Pattis did more than point out the problem of interpreting stepwise refinement solely as top-down design; he also devised an alternative interpretation.

In 1990, Pattis criticised traditional stepwise refinement and proposed *stepwise enhancement* as a more viable and useful alternative [Pattis 1990]. Pattis describe stepwise enhancement as follows:

First, students must reduce the program specification to a minimum, concentrating on the main structural features and ignoring all the complicated details that will make the program difficult to write. Then, students design, implement, and test a complete version of the program that meets the simplest specification. Then students proceed to the next stage, enhancing the specification to include some of the complicated details that were previously ignored. Once again, students design, implement, and test an enhanced version of the program, which meets the enhanced specification. The students continue repeating this process —at each stage enhancing the specification and writing an enhanced program that meets the specification— until the complete problem as described in the original specification has been solved.

Pattis argues: “Fundamentally the stepwise-enhancement technique is useful because it is easier to design, implement, and test a series of increasingly more sophisticated complete programs than it is to attempt writing one large program that solves the original problem specifications at the outset; that is, it is easier to solve a series of many small problems than it is to solve one big problem (commonly called ‘divide and conquer’). This technique also allows students to test their original ideas on how to solve the main features of the problem in a simple program first. They receive feedback, at very short intervals that tell them whether or not they are on the correct path to a solution program. So, if their initial ideas are incorrect, they can recognize this fact quickly and discard the ideas early in the programming process, without committing a lot of time and effort to pursuing them; such feedback is critical for students who are learning in parallel the language features and how to use these features when writing programs” [Pattis 1990].

Pattis continued his critique in [Pattis 1993], where he pointed out that it is not a good idea to apply stepwise refinement solely as top-down design; instead, beginning students should use model programs or other divide-and-conquer variants of stepwise refinement, e.g. prototyping or iterative development. Pattis wrote: “Iterative development allows programmers to establish landmarks on the path to a solution: they can test their code on realistic data at each of these landmarks (without stubbing) to evaluate their design; if it works, they can confidently proceed to the next landmark. Therefore, students can interleave the design and implementation phases, using each to check the quality of the other“. Pattis also pointed out that an early emphasis on design is misplaced; instead, students should start by coding programs designed by their instructor. We return to Pattis’ suggestion of stepwise enhancement later in this and in the following chapters.

Unfortunately, Pattis’ description of stepwise enhancement was not adopted by the community —perhaps because object-oriented programming entered the stage and, along with the eternal language debate, dominated the agenda.

6.2.4 From structured to object-oriented programming

Objects as programming entities were introduced in the 1960's in Simula, a programming language designed for making simulations, created by Dahl and Nygaard at the Norwegian Computing Centre in Oslo [Dahl et al. 1966].

During the 1980s, object-oriented programming became part of the curriculum at many universities —mostly due to Smalltalk [Goldberg et al. 1983, Ingalls 1978], C++ [Stroustrup 1985, Stroustrup 2000], and Eiffel [Meyer 1987, Meyer 1997]. In 1989, object-oriented programming in BETA became part of the introductory curriculum at the Department of Information Studies at University of Aarhus [Kristensen et al. 2007, Madsen et al. 1994].

Decker and Hirshfield were some of the first to make a case for teaching OOP in CS1 [Decker et al. 1992], but many others followed in the early 1990s [Berman 1996, Berman et al. 1994, Decker et al. 1993, Decker et al. 1994, Kölling et al. 1995, Luker 1994, Wallingford 1996, Willshire 1995, Wolz et al. 1994].

In the 1990s, the language debate was quite dominating and many languages were proposed as the first object-oriented (or object-based) language: Ada [Temte 1991], Smalltalk [Skublics et al. 1991], Turing [Holt 1994], C++ [Berman et al. 1994], Blue [Kölling et al. 1996a], and Java [Culwin 1997]. Brilliant and Wiseman provide an overview of the first programming paradigm and language dilemma in the post-Pascal era [Brilliant et al. 1996]. In the mid-1990s, C++ temporarily "won" the language war and became the dominating CS 1 programming language; in 1995 the College Board decided that C++ should supplant Pascal as the language for the AP program.[17] This most unfortunate decision was highly criticised by many prominent university professors, e.g. [Abelson et al. 1995]. Around the same time, Java entered the stage, and it quickly became the primary language of choice for introductory programming courses at universities. In 2000, only two years after the C++ decision about the AP program was implemented, the College Board decided that Java should supplant C++ in the AP program; that decision was implemented in 2003. Today, Java is by far the most widely used introductory programming language; in a survey conducted in the spring of 2004, Dale found that 65% of 148 responding educators were using Java and 25% were using C++. The remaining 10% were shared by languages such as Scheme, C, Ada, Python, and VB) [Dale 2005].

With the emergence of object-oriented programming languages in the introductory programming course, programming methodology virtually vanished from the programming education research agenda. It is reasonable to assume that the new paradigm and complex programming languages drew attention away from the methodological focus; educators seemed to be preoccupied with other concerns and dealing with more technological issues. Clearly, C++ is a very complicated language, but even Java makes people struggle; for example, due to the lack of simple I/O mechanisms in Java, many early textbooks on object-oriented programming in Java focused on graphical user interfaces!

Despite the fact that an increasing number of institutions are moving to adopt Java in their introductory curriculum, those institutions do not by any means report universal satisfaction with Java as a teaching language. The problems that arise in using Java at the introductory level were analyzed in more detail in a paper by Roberts [Roberts 2004a], and in early 2004, the ACM Education Board initiated the ACM Java Task Force to review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity [Roberts 2004b]. The result of the efforts of the Java Task Force is available online [Roberts et al. 2006].

[17] The advanced placement program, known as AP, offers high school students in the U.S.A. the opportunity to receive university credit for their work during high school. The AP program is administered by the College Board [Wikipedia 2007c].

The switch to teaching object-oriented programming has been treated more as a revolution than as an evolution; in doing so, important fundamental issues such as elements of programming methodology have been abandoned.

One of the few people who raised his voice on programming methodology and object-oriented programming was Meyer; under the headline *Design by Contract*, Meyer carried some of the key principles and techniques from structured programming to object-oriented programming [Meyer 1992, Mitchell et al. 2002]. However, Meyer also pointed to some of the differences between structured programming and object-oriented programming: "One of the differences involves the notion of top-down functional design, which, after the work of Wirth [Wirth 1971] and Mills [Mills 1971], has become almost uniformly identified with good software practice" [Meyer 1989]. Meyer argues that bottom-up design, where the programmer starts from available components and builds on them, is what characterises object-oriented programming. "Rather than attempting to produce the best possible solution for the current problem, you try to produce a good solution, minimizing the effort by building on previous achievements, and striving for the highest possible degree of generality to facilitate future developments" [Meyer 1989, p. 20]. Meyer point at another of the ideas and attitudes that emerged from the initial structured programming wave, the general hostility toward testing: "Tests were frowned upon in the structured programming literature, following Dijkstra's often quoted remark [Dahl et al. 1972, p. 6] that 'testing can never be used to show the absence of bugs, only to show their presence' with the understanding that only proofs will succeed in achieving the former goal" [Meyer 1989, p. 21].

Of course, the creators of structured programming did not really advocate full formal proofs but rather a general method of software production which, as described by Dijkstra [Dijkstra 1976] and Gries [Gries 1981], associates partly formal correctness arguments with the programs *as they are being built*. But, as in the case with stepwise refinement, the message was interpreted in a more narrow way than it deserved and due to that was mostly ignored.

Programming methodology has developed into the area known as formal methods. In that area, people meet biannually for a conference on Teaching Formal Methods [Dean et al. 2004]; however, the area has had little impact on CS education in general. The major, current methodological developments which influence educational practice come from expert consultants outside academia, e.g. Kent Beck, Ward Cunningham, Erich Gamma, Martin Fowler, Alistair Cockburn, and covers topics such as design patterns and frameworks, extreme programming, refactoring, agile development, and test-driven development [Beck 1999, Beck 2003, Cockburn 2002, Fowler 1999, Gamma 1995, Martin 2003]. See also section 4.2.6.

It would be a pleasure to see practitioners and theoreticians unite their forces and work more closely together for the development of theories, methods, tools and techniques for practical development of quality software; however, it does not seem likely that this unification will become reality in the foreseeable future.

6.2.5 Conclusion

Initially, there were many attempts at addressing programming methodology as problem solving, top-down design, and stepwise refinement in introductory programming education. However, stepwise refinement interpreted solely as top-down design is at best insufficient, at worst it does not work. Pattis suggested stepwise enhancement as a more viable and useful alternative to stepwise refinement. Although Pattis' idea seems more than promising, it never caught on in the programming education community —probably because object-oriented programming entered the stage and required the attention of educators for reasons other than methodological ones.

6.3 A future perspective

A necessary prerequisite of strict top-down stepwise refinement is that the programmer has to have the big plan from the outset. It can come as a vision, a revelation, by magic, or by some other means, but it has to be there. The conception that software can be developed in this way is fundamentally wrong. However, the imagination flourished for decades. It is the same imagination that drove the development of the waterfall methods of software engineering, and it is now realised that it was severely misleading in that area as well. Today, we know that software development does not work like that. It does not work for small systems, and it does not work for large systems either.

6.3.1 Best practice

In early 2001, motivated by the observation that software teams in many corporations were stuck in a quagmire of ever-increasing process, a group of industry experts met to outline the values and principles that allow software teams to work quickly and respond to change. They called themselves the *Agile Alliance* [AgileAlliance 2007]. Some of the principles of the Agile Alliance are: working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following *the* plan.

It is the ability to respond to change that often determines the success or failure of a software project. When we build plans, we need to make sure that our plans are flexible and ready to adapt to changes in the requirements specification. In [Cockburn 2002], the author discusses stability with respect to software development. Cockburn argues that program development starts in a situation of instability and that, over time, the amount of instability is reduced. Just prior to a design review, the work is relatively stable, but at that point reviewers and users provide new information that makes the work less stable again for a period. It is tempting to strive for maximum stability, and the simplest approach is to postpone design until requirements are stable and to postpone programming until design is stable —i.e. the waterfall approach to software development. However, this approach has a number of drawbacks. The first is that the elapsed time needed for the project is the straight sum of the times needed for requirements, design, programming, etc. The second problem is that surprises usually *do* crop up during the project (e.g. a supplier does not deliver on time, an algorithm does not scale as expected, and there are changes in the requirements); when it does, it causes *unforeseen* revisions of the requirements or design. The third and perhaps most severe problem is the absence of feedback from the downstream activities to the upstream activities. A different strategy —concurrent development— shortens the elapsed time, provides feedback opportunities, and allows seamless integration of changes in requirements and design.

We are not concerned with large-scale software development, but we are concerned with educating novices in programming skills that scale to situations where requirements are not fixed. Furthermore, the upstream feedback that concurrent development provides is even more important to novices as Pattis has pointed out (section 6.2.3).

6.3.2 A study of the programming practice of experts

To organize education for novices in the skills of programming, we need to know what to aim for. Best practice in industry, as described by expert consultants in object-oriented programming, provides part of the answer, but in order to get a closer look at the programming process of experts, we conducted a small qualitative study of experts undertaking an extempore programming assignment of an unfamiliar problem. Research of expert and novice programmers was conducted in the 1980s (see section 4.2.1), but we are not aware of recent studies of expert programmers.

Our aim was to conduct a small number of studies (five to ten) in order to get an impression of how details of the programming process of experts unfold over time when addressing a non-trivial and unknown programming task. We designed a programming problem that was easy to explain but still complicated enough to represent a challenge to experts; it should be possible to finish the task within three to four hours.

The problem was to develop software for the short message system of a cell phone (SMS). Using the numeric keyboard and a built-in dictionary of a cell phone, it is possible to write short text messages by pressing only one button for each character to be generated. In the cases where a sequence of digits has no corresponding word in the dictionary or when more than one word matches a sequence of digits, special action is needed. If a sequence of digits does not match a word, there is a special mode where it is possible to enter the word by pressing the digit keys repeatedly. A user-interface was provided, and so was a cell phone, which allowed the subjects to make experiments in order to fully understand the intended behaviour of the system. The problem called for a small but non-trivial state machine and a data structure for the dictionary.

Programming is a cognitive activity; therefore, understanding a person's programming process is a question of chasing cognitive structures —'a beetle in a box' to use Wittgenstein's example, in a box that can never be opened [Wittgenstein 1953]. Interviews have serious shortcomings in how they rely upon the reflective skill of the subjects; first, it is just too complex to get to describe what you do when you program; second, interviews also rely upon post-hoc justification of what the subject did rather than the in-situ practice of programming as pointed out in [Brown 2006]. A reflection after-the-fact will therefore be very superfluous and of little use for our purpose.

A general issue to consider for this kind of experiments is the so-called Hawthorne effect, i.e. the phenomenon that when people are observed in a study, their behaviour or performance temporarily changes [Wikipedia 2007a]. The Hawthorne effect is not a serious issue in this case since we are not particularly interested in the actual performance of the elected experts. Our ultimate goal is to identify ideal programming behaviour to aim for when planning and organizing programming education. Therefore, it is unproblematic if the subjects' performances in the experiments differ from a normal programming session. It is the essence of the performance that is of interest, not the details of effectiveness or precision.

We observed and video taped the subjects while they worked on the problem. Also, their computer screen as well as their voice was recorded during the experiment. In order to make the subjects reveal their thoughts during the process, we organized the experiments as pair-programming events. In two cases, the experiments were made with only one programmer; in both cases we interrupted and asked questions if the subjects forgot to think aloud. We conducted six experiments with nine subjects (three pairs and three individuals). Except for one subject, who was a PhD student, all subjects were professors at CS departments at research universities and experienced software developers. Five subjects were from Denmark, and four from the U.S.

The subjects addressed the problem in many different ways, but a number of techniques and principles were common to all or most of the experiments. One pair decided to follow a strict test-driven approach, and one pair spent almost all the time modeling, aiming at achieving a thorough understanding of the problem. One subject, the PhD student, approached the problem differently than the rest of the subjects, mixing different aspects of the task, and he did not get very far in the process; clearly, he was not as experienced a programmer as the other subjects. The remaining three experiments were very similar with respect to process as well as outcome. Except for the pair that pursued a strict test-driven approach that took a slightly different approach, all other subjects identified and addressed the essential part of the problem first. In all cases the rationale was: "once we have understood this, we are more or less done". Most of the subjects established understanding of the problem (and sub-problems) through programming experiments; only one pair used modeling activities at a white board to establish understanding of the problem. Thus, while the subjects had different approaches to establishing understanding, they all tried to establish understanding of the key aspects of the problem as quickly as possible.

Experiment 1: An interesting observation about the pair that followed a strict test-driven approach was that they had a well-defined set of techniques and principles and a corresponding vocabulary to guide them in their process, which in that respect became very foreseeable. But realizing the positive effect of applying the explicit techniques of test-driven programming was an interesting observation. Another interesting observation of this pair of subjects was their behaviour when they encountered a serious error. Lacking a corresponding set of techniques and terminology for the debugging process, their process became much more ad-hoc and less systematic in that phase.

Experiment 2: The pair that took a modeling approach embarked on the problem by discussing model issues and spent most of the time at a whiteboard constructing a static class model and a dynamic state model of the specified system. They tried to be very complete in the modeling, making sure that the model captured every aspect of the problem. When eventually they *did* start coding, it went fairly smoothly. We did not take the time to finish the system, and within the time frame we had reserved, the pair did not produce much working code.

Experiment 3: The PhD student addressed the problem in a very bottom-up fashion. This subject decided to address the problem of the dictionary and started considering various data structures for representing the dictionary. Only after approximately 60 minutes, and considerations of several alternative data structures and some sporadic coding, did he start worrying about the interface to the dictionary, i.e. how the dictionary would be used in the program. This subject did not get much further with the problem.

Experiment 4-6: One pair and two individual subjects were surprisingly similar in their approach to the problem. They all addressed the problem by applying the *solve-a-simpler-problem-first*-heuristic, i.e. by identifying or constructing simple part-problems that were solved one at a time. Of course, there were variations in the sequence of specific problems that were addressed, but overall they approached the problem in the same manner. As everybody else, they started to question the problem specification and try the cell phone in order to understand the exact problem specification. The next things all subjects did was to investigate the code that was provided as part of the problem specification.

In two of the experiments, the subjects first embarked upon the data structure for the dictionary, in the third experiment, the subjects assumed the existence of a dictionary with a specific interface and addressed client code that used the dictionary, i.e. code that implemented the state machine. None of the subjects modeled the state machine prior to programming it.

All subjects programmed incrementally or “sideways” (as opposed to top-down or bottom-up), i.e. they ignored most of the problem specification and addressed only a small part of the specification and developed code to implement that part; then they went on to address a new part of the original problem specification. In terms of the state machine, they focused upon one state at a time, and for that state considered only one entry transition at a time. In two of the experiments, the software was tested after almost every increment. In the third case, the software was tested ‘by need’, i.e. the subject tested only when he felt the need for it. The subject that started with the data structure for the dictionary did so by choosing a standard data structure from the API, realizing that it might not be an optimal choice with respect to time and space but that it surely would be optimal with respect to programmer resources.

Three of the four subjects (two of the three experiments) deliberately decided to produce a quick and dirty solution; one subject explicitly said that he always plans to produce two solutions. The first is made to fully understand the problem and get experience with one set of ideas for solving the problem. This subject also ‘admitted’ to producing poorly structured code in the first solution. After the experiment, this subject said, “While my first solutions very well might work, they are typically very coarse-grained in the class structure and need a lot of refactoring in order to survive as candidates for second solutions. Many of my programs end up with a lot of ‘global’ variables, i.e. I typically have very complex representation invariants for my classes; however, I still think of the code in ‘equivalent classes’ so to speak —I just don’t waste time on premature refactoring because I know I have a second shot”. In other words: when making the first solution, this subject sacrifices refactoring for the benefit of quick progress, realizing that a decent factorization is one of the primary concerns of the second solution, which is created only when the problem is properly understood and ideas for the solution have been exploited.

While the solutions that were produced were relatively poorly structured according to standard measures, the subjects still applied systematic techniques such as (implicit) class invariants to capture the purpose of the variables in a class. The notion of class invariant was only mentioned explicitly by one subject, but most of the subjects applied the technique “silently”; when the problem specification was enhanced (i.e. a new part-problem considered) and extra behaviour was needed in order to fulfil the specification, the subjects systematically ensured that the implicit class invariant was maintained by updating all relevant variables —wherever they were declared and whether encapsulation was applied or not. When confronted with this behaviour afterward, the subjects explained their behaviour with reference to the aforementioned principle of always making two solutions and deferring decent factorization and modularization to the second round.

All subjects progressed in two fundamentally different ways. In cases where the part-problem under consideration was well understood at once, and a solution immediately came to mind, a classical strict top-down or bottom-up approach was applied. However, in cases where the problem currently under consideration was not fully understood, the programming efforts of the subjects are best characterized as a discovering, opportunistic activity not determined in advance. This kind of activity involves making small experiments to verify hypotheses of what could be in the efforts of conceiving a solution. Not all of these hypotheses needed being implemented to be rejected; due to the subjects’ level of expertise, many flawed hypotheses were discarded without writing any code.

Robillard has studied human behaviour in software engineering [Robillard 2005]. Robillard found a similar behaviour in his studies: “Experienced software engineers and programmers often don’t need a lot of detail to understand the task to be done, so they can adopt a systematic or breadth-first approach. In contrast, novices often rely on their own understanding of detailed programming language statements. They thus adopt an opportunistic or depth-first approach —they must go to the detail level to get enough information to understand the task to be done. Any engineering activity that involves some creativity will result in a mixture of opportunistic and systematic approaches [...]. Both systematic and opportunistic cognitive actions are often coupled with problem-solving activities, so software engineering practices should take this into consideration and provide guidelines to support both behaviours” [Robillard 2005, p. 63]. Winslow made a similar observation when studying expert and novice programmers: “Experts, when given a task in a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general (opportunistic) problem solving” [Winslow 1996, p. 18].

The overall conclusion of the six experiments is that almost all the experts proceeded according to a strategy that we may characterize as *solve a simpler problem first*. Some subjects planned on producing two solutions: the first to understand and the second to produce a teachable/comprehensible structure. The subjects practised strict top-down or bottom-up programming only locally (i.e. in the individual increments of the overall strategy) and only in the cases where the problem was well understood and the solution was obvious; in all other situations, their behaviour is best characterised as an explorative activity of discovery and invention in search for a solution to the problem under consideration.

It is very rare (if ever) that novices are in a situation where the problem is well understood and the solution is obvious; it is therefore a natural consequence to identify and teach other programming skills and techniques than the classical top-down or bottom-up interpretations of stepwise refinement.

6.3.3 Horizontal programming

Adhering to the principles of agile development, and in concordance with the programming practice of experts as demonstrated by our small study described in the previous subsection, we abandon the idea of stepwise refinement as strict top-down programming as it is illustrated in Figure 6-1.

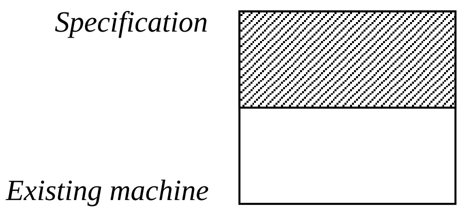


Figure 6-1: *Strict top-down programming*

The obvious alternative to top-down programming, and the one promoted in the early days of object-oriented programming by Meyer [Meyer 1989] and others, is bottom-up programming as illustrated in Figure 6-2.

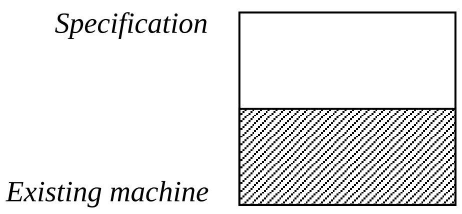


Figure 6-2: *Strict bottom-up programming*

Bottom-up programming is much more realistic than top-down programming, and it offers many of the qualities praised by Pattis in his description of stepwise enhancement (e.g. design, implementation, and test of a series of many small problems as well as frequent feedback). However, strict bottom-up programming, as described by Figure 6-2, is based upon the assumption that the total specification is known from the outset and thus does not accommodate (seamless integration of) changes in requirements.

Instead, we suggest an approach to program development similar to what Pattis describes as stepwise enhancement. Since we consider it an alternative to the vertical variants of top-down and bottom-up programming, and for lack of a better term, we denote it *horizontal programming* (see Figure 6-3).

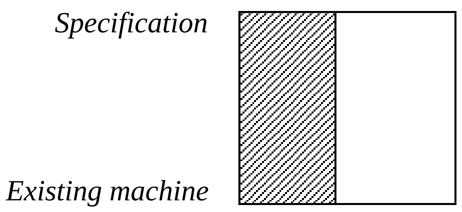


Figure 6-3: *Horizontal programming*

Of course, top-down and bottom-up programming do take place in the small as subprocesses within the major refinement process, but it is sensible to think of the overall progression of the refinement process as horizontal rather than vertical. There are several reasons for this, some of which was mentioned in section 6.2.3 in our description of stepwise enhancement and some of which was mentioned in the previous section on experts' programming process: (1) this view provides a seamless transition from initial development to further development or maintenance, i.e. incorporation of new requirements; (2) practicing programming this way provides instant feedback, which is extremely important to novices in a learning process; (3) it is easier to design, implement, and test a series of increasingly more sophisticated complete programs than it is to attempt to write one large program that solves the original problem specifications at the outset.

From a learning-theoretic point of view, the horizontal approach is also better because it smoothly integrates with *worked examples*, *example-completion*, and the principle of *faded guidance* as described in section 3.3. This aspect is illustrated in Figure 6-4; the black area indicates what is provided as worked examples by the instructor —over time, the students must solve more and more of the problem and eventually even provide the specifications of new requirements that are added to the initial problem specification.

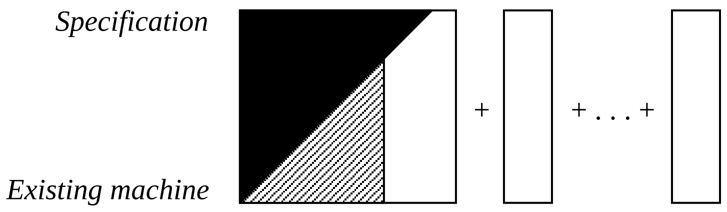


Figure 6-4: *Faded guidance, horizontal programming, and new requirements*

Sweller's research program in cognitive load theory (see section 3.3) accumulated empirical evidence showing that traditional, practice-based problem solving was less than an ideal method for improving problem-solving performance when compared to instruction based upon worked examples, example completion and faded guidance. As mentioned in section 3.3, the worked examples literature is particularly relevant to programs of instruction that seek to promote skill acquisition, e.g. music, chess, and programming because learning from worked examples causes learners to develop knowledge structures representing important, early foundations for understanding and using the domain ideas that are illustrated and emphasized by the instructional examples provided. These representations guide problem solving, and they may be conceptualized as representing early stages in domain schema development and in the acquisition of expertise.

For learning-theoretic reasons, we therefore adopt an approach to programming education where novices first do very simple tasks, e.g. modify a method or add a method similar to an already existing method in a class. From there, they go on to implement complete methods for stubs and specifications which are provided. Then they "graduate" to adding complete methods and specifications on their own but to existing classes. Students write only small bits of code; they do not yet design classes. Class design is indeed very hard; coding from specifications is not trivial for novices, but it is orders of magnitudes simpler than doing class design. Later again, new requirements to an existing project are provided, and students design a small number of new classes to represent the key concepts from the new requirements. Eventually, after half a year or so, students are ready to design and implement small systems from scratch. As indicated above, this approach smoothly integrates with horizontal programming.

There are other reasons for our approach than the learning-theoretic arguments just provided. In practical software engineering, systems are rarely developed from scratch; they are grown. Thus, our approach to programming education resembles common programming practice.

Our approach is different from traditional approaches, but there are good reasons for it —practical as well as learning-theoretic— and our experiences demonstrate that it works extremely well.

6.4 Conclusion

This chapter has provided answers for the two first parts of research question four: $Q_{4.1}$: How has programming methodology influenced programming education in the past? and $Q_{4.2}$: How can we characterize best-practice of modern software development?

In the 1970s and 1980s, early development in programming methodology (divide-and-conquer, top-down design and stepwise refinement) had a significant impact on programming education and programming education research. However, in spite of warnings from Wirth and Denning, stepwise refinement and structured programming was primarily interpreted as strict top-down design and programming.

With the emergence of object-oriented programming in the early 1990s, programming methodology vanished from the agenda of the programming education research community. It is reasonable to assume that the new paradigm and complex languages drew attention away from the methodological focus; educators seemed to be preoccupied with other concerns and dealing with more technological issues.

Pattis' suggestion of stepwise enhancement as an alternative to the traditional interpretation of stepwise refinement seems to better match best-practice of modern software development; however, Pattis' suggestion was not picked up by the community.

We are concerned with educating novices in programming skills that scale to situations where requirements are not fixed and correspond to best practice in industrial software engineering. Our study of the programming process of experts —which are confirmed by other studies — has revealed that experts primarily progress according to a horizontal solve-a-simpler-problem-first-strategy similar to Pattis' stepwise enhancement. In the process, programmers apply opportunistic problem-solving, and the programming process is best characterised as an explorative activity of discovery and invention. Vertical approaches are only applied in simple and trivial situations —the kind of situations that by definition never occur to novices. We therefore abandon vertical approaches to programming education and instead focus on a horizontal approach that corresponds to best practice and at the same time has obvious educational advantages.

7 Stepwise Improvement

In this chapter, we shall address the remaining parts of research question 4:

$Q_{4.3}$: How does best-practice in modern software development relate to programming methodology?

$Q_{4.4}$: Can we provide a characterization of the programming process that unifies programming methodology and best-practice of modern software development?

In the previous chapter, we argued that programming methodology, best practice, and programming education research have drifted apart. In this chapter, we propose a conceptual framework that indicates a potential (re-) unification of research in programming methodology, best practice, and programming education research. From our perspective, the key common denominator is specifications —formal or informal expressions of the intended behaviour of a program.

Assertions and specifications play a key role in the sub-area of programming methodology known as the *refinement calculus*; assertions also play a key role in Meyer's *design by contract* perspective on object-oriented programming; finally, specifications —in the form of (assertions in) test cases and test suites— play a key role in the specific best practice form of software development known as *test-driven development*.

Specifications may be expressed in many ways, e.g. as formal specifications in predicate calculus or some special specification language, as informal specification expressed in (more or less structured) natural language, or as a set of test cases. The exact form of specifications is not important to us; it is the notion of specification itself, the possibility of interpreting specifications as contracts, and the role of specifications in the programming process that is our concern.

Despite the variety of methodological approaches and their diverse degrees of formalisation, it is possible to unify the methodologies in a conceptual framework with specifications as the key common denominator. Section 7.1 presents the conceptual framework that —although inspired by object-oriented programming— is independent of any particular programming paradigm. Section 7.2 provides a discussion on the nature and role of specifications, programming activities and programming techniques at various levels of abstraction in object-oriented programming. Section 7.3 concludes the chapter.

7.1 Toward a unified programming methodology

This section is about programming methodology, the study of methods for making programs. The particular task we address is the systematic, horizontal development of programs from their specifications. For this task, we provide a uniform model that captures a variety of programming methods of varying degrees of formality, feasibility, and scalability.

Our model is macroscopic in the sense that the atomic items are *mechanisms* and their *specifications*. Our concern is neither interior structure nor syntactic form of mechanisms and specifications; our concern is *how mechanisms come into existence* from their specifications. Our concern is the *process* of development, not the *structure* or *syntax* of the developed.

The model is a conceptual framework, a system of well-defined terminology, that enables discussions about and characterization of programming methods in general, independently of the premises of specific methods and programming paradigms. Special emphasis is put on concepts for the characterization of program development as a goal-directed activity targeting a contract or specification that describes the intended behaviour of the program to be developed.

The conceptual framework can serve as an aid in gaining a coherent understanding of programming methodology that can be exploited for the development of that area (theories, methods, guidelines, languages, development tools, etc.) as well as for the development of programming education. The latter opportunity is exploited in chapter 8, where we present a programming method for novices and in chapter 9, where we discuss implications and opportunities for the instructional design of programming education based on the model for incremental development developed in this chapter.

We begin our endeavour with an example of a programming method with a rigorous, formal foundation: the refinement calculus. As pointed out in the previous chapter, the method is of limited practical feasibility and, consequently, does not scale well to larger programming tasks. However, it provides a basic terminology applicable for our purpose.

7.1.1 The refinement calculus

The refinement calculus is a formal approach to stepwise refinement for program construction [Back 1978, Morgan 1994, Morgan et al. 1992, Morris 1987]. It is a theory and a set of rules for deriving imperative programs from their specifications.

A fundamental idea is to regard specifications as programs but to distinguish between *abstract programs* and *concrete programs*, i.e. non-executable programs and executable programs. In short, the refinement calculus is a theory of programming based upon *behaviour-preserving* program transformations from abstract to concrete programs. (The emphasis on behaviour-preserving transformations more than anything indicates that the programmer has to have to big plan from the outset as discussed in the previous chapter.)

The refinement process is about transforming abstract code (specifications) to concrete code while preserving the behaviour of the program. Fundamental to the calculus is the refinement relation $\leq$, which is a relation between programs. $P \leq Q$ means that program Q is better than program P in the sense of being less abstract. The relation $\leq$, between programs, is called *refinement*, and we say that Q refines P.

If *p* is a program, *p.concrete* expresses that *p* contains only executable code, and *p.refine* denotes a behaviour-preserving transformation that reduces (does not increase) the amount of abstract code in *p*.

In algorithmic form, and with a few notational liberties, we can describe the programming process as the transitive closure of refinement steps. The process is described algorithmically in Schema 7-1.[18]

```
$p := spec$ ;
{ **inv** $spec \leq p$ }
**do** $\neg p.concrete \rightarrow p.refine$ **od**
{ $p.concrete \wedge spec \leq p$ }
```

Schema 7-1: *Program refinement*

This is a special case of our model (to be developed in section 7.1.2) in the sense that the extra requirement we have in our model, that a program be total (i.e. fulfils the requirements), is trivially true in the case of program refinement.

Provided that *spec* is computable, the process described in Schema 7-1 is partially correct; if (patiently) applied and if it terminates, it will lead to a concrete program that satisfies its specification. However, as demonstrated in section 6.3 there is little evidence, to say the least, that this is the way experts develop programs let alone *could* develop programs.

There are indeed many useful techniques in the refinement calculus, and refinement as such plays an important role in software development; however, it is not rich enough to count as a general model of systematic development of programs from their specifications. Not even close! However, the basic notation and terminology from the refinement calculus is useful; we will use it and extend it for our purpose.

7.1.2 A conceptual framework for program extension

Our interest is not in behaviour-preserving program transformations from abstract to concrete programs. On the contrary, our interest is in *behaviour-improving* program transformations from partial to total programs (the terminology is precisely defined later in this section). Our perspective is orthogonal to the existing refinement calculus, i.e. we extend the classical refinement theory in a new, independent dimension. The refinement calculus operates along the abstract-concrete dimension of programs (moving from abstract to concrete); our extension operates along the partial-total dimension of programs (moving from partial to total).

The principle of refinement —specify today, implement tomorrow— is well-known and supports separation of ‘what’ from ‘how’. The principle of incremental development —solve a simpler problem first— is less well-

[18] We denote these algorithmic descriptions as schemas since they are schemas of cognitive processes applied by programmers in the process of program development; they are not meant to be executed by machines but to be applied by people (see section 3.1).

known, but frequently practised. It is one of the many tacit competences of experts and supports the creation of simpler ‘what’s. We use it all the time in programming and in all other aspects of life for that matter —not least in research! Solving a simpler problem is realised by ignoring for a while parts of the requirements and only address a smaller part of the problem. Gradually, an increasingly larger part of the requirements is considered until eventually the original problem is solved. The reason for applying the principle is always to get things going, make some progress, learn more about the problem and eventually fully understand it, and get ideas for (the structure of) a solution to the problem. It is an incarnation of one of Dijkstra’s mantras: separation of concern. It is a superb way of mastering complexity and one of our primary instruments of thought.

If we were to build a traffic warning system aimed at providing early warning to vehicles about road conditions, such as whether the road is slippery, we would break the problem down into a number of sub-problems to be investigated separately and for each sub-problem we would first address a very simple variant of the problem and then, gradually, improve our solution to address an increasingly larger part of the requirements. One sub-problem is data collection; another is data distribution. We want cars to exchange information when passing each other on the road; thus, one sub-problem concerns how to provide safe and efficient communication between moving vehicles. Regarding data collection, we might start out with initial experiments on data collection by building a sensor prototype and test it in a controlled test environment, say a freezer with a black, spinning plate simulating the hard top of a road and the movement of a car. Later, when the basic technology for data collection is in place, we might want to test it in the real environment and conduct experiments with cars driving around to test the prototype under various conditions.

A more trivial example of solving a simpler problem first and then gradually improve the solution to finally cover the full set of requirements is the construction of a program to simulate a calculator. For a calculator program, we could for a while ignore multi-digit numbers, operators, precedence rules, and decimal-point numbers. In small increments we could add more and more functionality to the program, and eventually end up with a program meeting the full set of requirements.

In the following we define a taxonomy that allows us to be more precise about key aspects of the process of incremental development of mechanisms from their requirements.[19] Specifications play a central role as already indicated.

Definition 1: A *mechanism* has two *specifications* and an *implementation*. One specification, which we denote by *req*, or just *r*, describes the *ultimate requirements* of the mechanism; the other specification, which we denote by *spec*, or just *s*, describes the *specification* (intended behaviour) of the current implementation of the mechanism. The implementation is denoted by *impl*, or just *i*. For mechanism *m* we refer to the two specifications as *m.r* and *m.s* and to the implementation as *m.i*.

[19] Developing programs from their requirements does not mean that the full set of requirements must be available before the programming process begins. The process we propose accommodates seamless integration of new requirements.

Starting with the requirements m.r, we want to develop a specification m.s that implies m.r and an implementation m.i that meets specification m.s. But as the development process proceeds, the specification may not imply the requirements and the implementation may not meet the specification. The slack in two places gives room for incremental development in two ways: improve the specification to better match the requirements and improve the implementation to better match the specification. The refinement calculus is concerned only with the latter.

A non-deterministic description of the incremental process we propose for constructing a mechanism from its requirements can be sketched as a process of two (or more) independent activities. In Schema 7-2, the first guarded command models repeated improvements of the specification and the second guarded command models refinement of the implementation to meet the current specification. A new guarded command is added for each additional activity we may choose to include in our model of the programming process (e.g. refactoring and optimization).

```
spec:= the empty specification ;
do spec does not imply req → improve spec
 [] impl does not meet spec → improve impl
od
```

Schema 7-2: *Sketch of stepwise improvement*

In the refinement calculus, there is no distinction between *spec* and *req*, i.e. there is only on specification; in this situation, the programming process degenerate to the second guarded of Schema 7-2 (as described in Schema 7-1).

In the remaining part of this chapter, we ignore the second guarded command and concentrate on the first, i.e. we concentrate on aspects of stepwise improvement of the specification of a mechanism toward its requirements.

Specifications

The programming process we envisage is characterised by a sequence of intermediate specifications, s_0, s_1, s_2, ..., s_n where s_0 is the initial specification and s_n is (or implies) the ultimate requirements. The intermediate specifications correspond to the sequence of values held by *spec* during iterations of the first guarded command of Schema 7-2.

Definition 2: A *specification* is a description of the (intended) behaviour of a mechanism or an implementation. It takes the form of a contract with mutual obligations between a *client* of a mechanism and the mechanism and is expressed in two components called the precondition and the postcondition respectively. For specification *x*, we denote the two components by *x.pre* and *x.post*.

For example, a specification of a mechanism may express that it backs up 1 Gb/sec (postcondition) provided there are at least 40 Gb free disk space (precondition).

A concrete specification may be expressed in many ways; it may be a formal specification expressed in predicate calculus or some special specification language, or it may be an informal specification expressed in (more or less structured) natural language or in pictures; it may be provided as a set of test cases; it may even be implicit, based upon naming conventions and/or friendly interpretation of names of mechanisms. The exact form of specifications is not important to us; what matters is the notion of specification itself, the possibility of interpreting specifications as contracts, and the role of specifications in the programming process.

The notion of mechanism may be used to describe many things. In the context of object-oriented programming, one may think of a mechanism as a generalisation of method and class. However, the notion of mechanism is more general and allows a multitude of interpretations across programming language paradigms. Consequently, the conceptual framework we are going to present provides a uniform model that captures a variety of programming methods across language paradigms.

We take the client's view in describing and discussing specifications. Relation $S \leq T$ means that specification T is better than specification S, i.e. T makes more requirements than S. The relation $\leq$, between specifications, is called *extension*, and we say that *T extends S* and *S reduces T*. We also say that T is *stronger* than S and that S is *weaker* than T.

Definition 3: *T extends S* (and *S reduces T*) is written $S \leq T$ where

$$S \leq T \;\equiv\; S.pre \Rightarrow T.pre \;\wedge\; T.post \Rightarrow S.post$$

In plain English, a specification is extended by increasing the requirements, i.e. weakening the precondition or strengthening the postcondition. Weakening the precondition means increasing the number of initial cases for which the specification is defined; strengthening the postcondition means reducing the set of possible results.

For example, a specification that a watch is splash-proof is weaker (i.e. worse for the client) than a specification that it is water resistant to 100 meters, and a specification that a piece of software requires at least 30 Mb is stronger (i.e. better for the client) than a specification that it requires at least 60 Mb.

Definition 4: For notational convenience, we write $S = T$, $S \neq T$, and $S < T$ with obvious semantics; in the latter case, we say that T is a *strict* extension of S (and S is a strict reduction of T).

Definition 5: Two specifications that are not related by the relational operator $\leq$ are *incomparable*. Incomparability of specifications is denoted by the relational operator #:

$$S \# T \;\equiv\; \neg(S \leq T) \wedge \neg(S \geq T)$$

The specification that a book costs \$20 in paperback and \$300 in hardback is not related to the specification that the lift capacity is 750 kg. Similarly, the specification that the water temperature is at least 24° Celsius in July is not related to the specification that the water temperature is at least 10° Celsius in December.

Remark: We express weakening and strengthening of conditions as logical implications because basically that is what it is. However, we do not mean to suggest that specifications need to be expressed formally; the model is applicable regardless of the concrete expression of specifications. *End of remark.*

Remark: Despite the notational and semantic uniformity, it is expedient for a while to distinguish our use of ≤ from its use in the refinement calculus. Our definition of the operator is the same as in the refinement calculus, but we use it for a different purpose: extension rather than refinement. We change the specification and modify the implementation accordingly; in the refinement calculus the specification is kept constant and the program is made more concrete. Later we shall unify the two theories. *End of remark.*

Definition 6: A specification with precondition Q and postcondition R is written as $[Q, R]$.

The *weakest specification* of all is [false, true]. From a client's perspective, it is the worst specification of all, for it is not guaranteed to operate in any state (precondition false), and if it does, it has complete freedom to do anything (postcondition true). Similarly, the *strongest specification* of all is [true, false]. From a clients perspective it is the best specification of all; a mechanism with specification [true, false] always operates (precondition true) and establishes the impossible (false).

Definition 7: The extreme specifications *abort* and *miracle* are defined as:

$$\begin{aligned} \mathit{abort} &\equiv [\text{false, true}] \\ \mathit{miracle} &\equiv [\text{true, false}] \end{aligned}$$

It is trivial to build a mechanism with specification *abort* and impossible to build a mechanism with specification *miracle*. Still, the extreme specifications play a role in iterative and incremental implementation of mechanisms (e.g. *abort* is a trivial first value of *spec* in its incremental approximation of the ultimate requirements *req* of a mechanism).

The set of specifications equipped with the relational extend operator forms a complete lattice with meet (∧) and join (∨) defined as

$$\begin{aligned} S \wedge T &\equiv [\ S.pre \vee T.pre\ ,\ S.post \wedge T.post\ ] \\ S \vee T &\equiv [\ S.pre \wedge T.pre\ ,\ T.post \vee S.post\ ] \end{aligned}$$

$S \wedge T$ describes a mechanism that behaves as S and T, and $S \vee$ T describes a mechanism that behaves as S or T.

abort is the bottom element and the unit with respect to the join operator; *miracle* is the top element and the unit with respect to the meet operator (join and meet is standard lattice terminology [Davey et al. 2002]).

The lattice of specifications can be sketched as in Figure 7-1 with the strongest specification at the top and the weakest at the bottom.

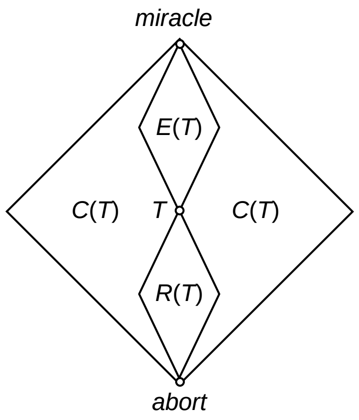


Figure 7-1: *The lattice of specifications partitioned by T*

A specification *T* divides the complete lattice of specifications into three subsets *E*(*T*), *R*(*T*), and *C*(*T*) characterized as:

$E(T)$	$=$	$\{ S \mid S \geq T \}$	*extension set*
$R(T)$	$=$	$\{ S \mid S \leq T \}$	*reduction set*
$C(T)$	$=$	$\{ S \mid S \# T \}$	*crossover set*

E(*T*), the set of stronger specifications, is called the extension set of *T*. *R*(*T*), the set of weaker specifications, is called the reduction set of *T*. *C*(*T*), the set of incomparable specifications, is called the crossover set of *T*. The extension set and the reduction set are also lattices.

For the characterization of specification lattices, we introduce the notion of *span*.

Definition 8: For two specifications *S* and *T* such that $S \leq T$, we define the *span* of *S* and *T* as:

$$\langle S, T\rangle \;\equiv\; \{ X \mid S \leq X \leq T \}$$

The span of two specifications is itself a complete lattice. Using the *span* notation, we can characterize the *universe,* the extension set, and the reduction set of a specification *S* as:

⟨*abort*, *miracle*⟩	the complete lattice of all specifications, the *universe*
⟨*abort*, *S*⟩	the reduction set of *S*
⟨*S*, *miracle*⟩	the extension set of *S*
⟨*S*, *T*⟩	the extension set of *S* targeting *T* and the reduction set of *T* targeting *S*

Totality, extension, and development traces

As previously mentioned, our concern is the process a programmer applies to systematically develop a program from its specification. We will *not* pretend that programmers are capable of instantly writing code that satisfies the requirements specification —not even for a single method. The game we will play is to model *incremental development of mechanisms*.

For a moment, we assume that a mechanism's implementation meets its specification and concentrate on the potential slack between specification and requirements. The notions of *total* and *partial* capture this aspect of mechanisms.

Definition 9: A mechanism *m* is *total* if the specification of its implementation is stronger than (or the same as) the requirements. A mechanism is *partial* if it is not total.

$$\begin{aligned} m.total &\equiv m.s \geq m.r \\ m.partial &\equiv \neg\, m.total \end{aligned}$$

For example, if the requirements of a mechanism is that it backs up 1Gb/sec provided there are at least 40 Gb free disk space, and we have build a mechanism that backs up 2 GB/sec provided there are at least 30 Gb free disk space, then the mechanism is total. If on the other hand the current specification is that the mechanism requires 60 Gb free disk space and backs up 1Gb/sec, then it is partial.

Figure 7-2 is a visualization of a mechanism —a so-called *box view*. The outer (white) box describes the mechanism. The borderline of the mechanism describes its specification, *r*. The inner (grey) box describes the implementation of the mechanism. The borderline of the implementation denotes its specification, *s*. The white area between *s* and *r* is called the *slack*. The slack can informally be characterized as *r – s*.

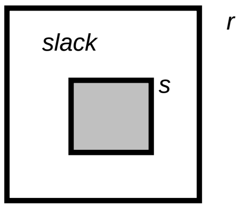


Figure 7-2: *Box view of a mechanism during incremental development*

Incremental development of a mechanism means that a mechanism over time (due to behaviour-altering transformations) may have different implementations; an early implementation fulfils a small part of the requirements while a later implementation fulfils a larger part of the requirements. To capture the notion of incremental development, we define a *development trace* as follows:

Definition 10: The *development trace* of a mechanism *m* is a sequence of specifications s_i, $0 \leq i \leq n$, where s_0 = *abort* and $s_n \geq r$.

It follows from the definition that a development trace has length at least 2.

Figure 7-3 visualizes a *subsequence* (of length six) of the development trace of a mechanism. Each specification of the trace is visualized in the context of the mechanism being developed (the previous specifications are shown as dashed boxes).

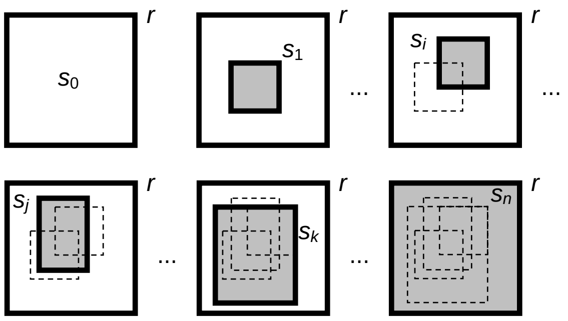


Figure 7-3: *Subsequence of the development trace of a mechanism*

Example: Development of a *setToNextDay* method of a class representing a date may exhibit the following development trace:

r: [true, this is the next date according to the Gregorian calendar]

s_0: the method does nothing

s_1-s_5: the method gives the correct result *provided*
s_1: the day is not the last of the month
s_2: there are 30 days in every month
s_3: the day is not 28 February of a leap year
s_4: the day is not 28 February of a century
s_5: the day is not 28 February of a 4-century

s_6: *r*

End of example.

The goal of any development process is *as smoothly as possible* to get from s_0 to s_n. Some development traces are more optimal than others; the ideal is a *monotone* development trace, like the one sketched above.

Definition 11: A development trace s_i, $0 \leq i \leq n$, is *monotone* iff it is a sequence of extensions, i.e. ($\forall\ i \mid 0 \leq i < n : s_i \leq s_{i+1}$). Similarly, a development trace is strict monotone iff it is a sequence of strict extensions (<).

Figure 7-4 shows two alternative visualizations of a monotone development trace of a mechanism, the box view and the lattice view.

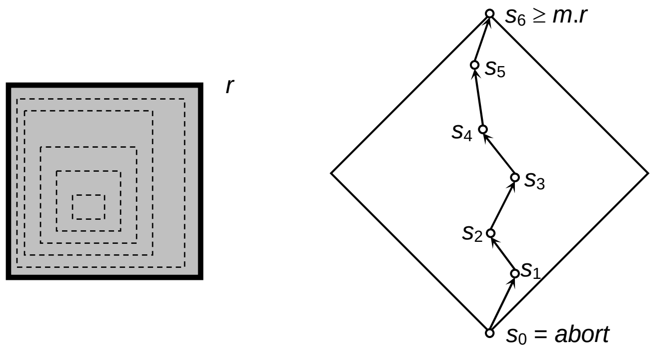


Figure 7-4: *Box view and lattice view of a monotone development trace*

In terms of lattice theory, a monotone development trace of mechanism *m* is a monotone path from the weakest specification, *abort*, to *m.r* or, more precisely, to a specification s_n that implies *m.r*.

In practice, unfortunately, monotone development traces are rare. Often one gets *stuck* in the sense of not being able to develop an implementation for an extension of the current specification. If one gets stuck, a *reduction* or a *crossover* (a reduction followed by an extension) is the only option.

Remark: Figure 7-5 provides a complete characterization of the possibilities of how one can proceed from a given stage in a development process for the situation where the current stage satisfies $s_k \leq r$. The current stage in the development process is described by specification s_k, which defines three disjoint sets of specifications that exhaust the options for the next stage: (1) extend to a stronger specification, (2) reduce to a weaker specification, and (3) crossover to an incomparable specification. The first case has three subcases; the second has two.

1. *Extent* to a stronger specification: $s_{k+1} > s_k$.
   a. Extend to s_{k+1} where $s_{k+1} \geq r$. In this case, the mechanism is total.
   b. Extend to s_{k+1} where $s_k < s_{k+1} < r$. In this case, the mechanism is not yet total, but we are on target.
   c. Extend to s_{k+1} where $s_k < s_{k+1} < miracle \ \wedge \ s_{k+1} \# r$. In this case, we have a better mechanism, but we are off the specific target *r*. However, this may be a smoother way to a solution, i.e. a specification in the extension set of *r*.
2. *Reduce* to a weaker specification: $s_{k+1} < s_k$.
   a. Backtrack to the previous specification s_{k-1}, i.e. $s_{k+1} = s_{k-1}$.
   b. Reduce to s_{k+1} where $s_{k+1} < s_k$.
3. *Crossover* to an incomparable specification: $s_{k+1} \# s_k$.

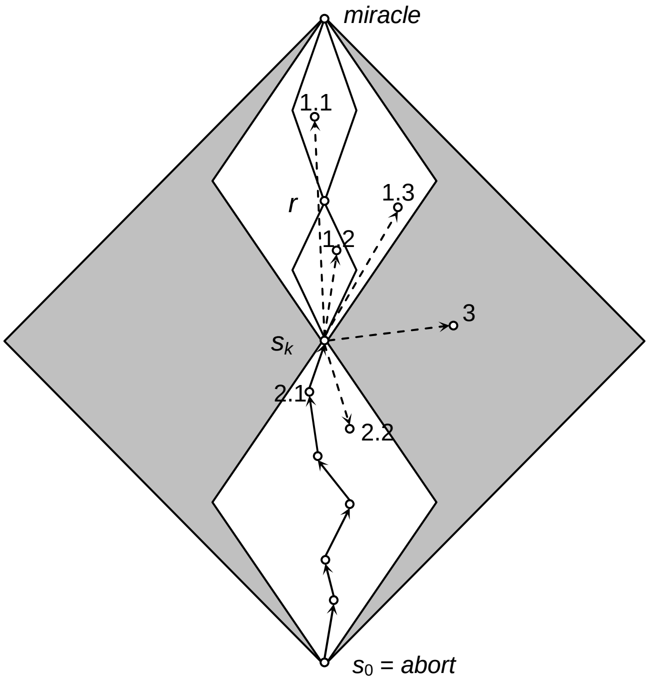


Figure 7-5: *Extension, reduction, and crossing over*

A similar enumeration can be made for the case where we already are off target, i.e. where $\neg(s_k \leq r)$. However, it is not the specific details of these enumerations that are important; the point is to emphasize that the general non-linearity of the programming process is captured by the notion of non-monotone development traces that are precisely defined in terms of specifications that describe the behaviour of the current version of the program under development. *End of remark*.

In the remaining part of the dissertation, we restrict ourselves to incremental development processes with monotone development traces. We do so not because it best captures realistic program development scenarios —on the contrary— but because it is sufficient for the way we intend to apply the theory in our discussion of instructional design for novices.

It is important to note that incremental development of mechanisms, monotone or not, copes seamlessly with new requirements. New requirements just mean that the finish line moves further away and, consequently, that we have to perform more increments and that the development trace becomes longer. It is still the same game we are playing. Because the current specification (s_k) is between the implementation (i) and the requirements (r), nothing is changed from the programmer's perspective by changing the requirements. It is always *programming toward a moving target*.

Consistency and refinement

We now turn to the situation where a mechanism's implementation not necessarily meets its specification. This situation introduces another kind of slack, namely between implementation and specification; the notion of consistency captures this aspect of mechanisms.[20]

Definition 12: A mechanism *m* is *consistent* if its implementation is stronger than (or the same as) its specification, i.e. if the implementation behaves according to its specification and *inconsistent* if it is not consistent:

$$m.consistent \equiv m.i \geq m.s$$
$$m.inconsistent \equiv \neg m.consistent$$

A mechanism where the specification (*s*) is provided as a set of test cases and where the implementation (*i*) satisfies the test cases is consistent. The mechanism may or may not be total; that depends on how well the test cases cover the requirements (*r*).

An improvement of a mechanism's behaviour with respect to its specification is called a *refinement*. Figure 7-6 describes refinement of the implementation of a mechanism for fixed specification and requirements. The green and red areas of the inner box describe how well the implementation meets the current specification. In the case where the current specification is provided as a set of test cases and unit testing is applied, the red area corresponds to red bar and the green area to green bar [Beck 2003].

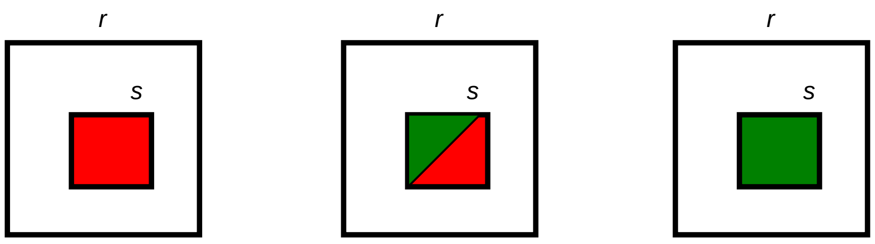


Figure 7-6: *Box view of mechanism during refinement of its implementation*

7.1.3 Unification of methodologies

In this section, we present a process for iterative, incremental development of mechanisms based on transformations of the specification of mechanisms and, consequently, transformations of implementations of mechanisms. The approach embraces the refinement calculus, and there is strong evidence that it is superior to the traditional refinement calculus with respect to capturing expert programmer's behaviour and best practice in general.

As mentioned in the previous section, we restrict attention to incremental development processes with monotone development traces; we consider only extensions of specifications, not reductions or crossovers. To enable a precise (schematic) expression of the notion of incremental development, we define a bit of notation:

[20] To enable expressions of relations between implementations and specifications, we expand the domain of the ≤ operator to cover implementations as well as specifications. (This is similar to the refinement calculus where specifications are considered to be programs and the operator is defined between programs.)

Definition 13: Specification extension of a mechanism is denoted by the imperative *m.extend* and implementation refinement of a mechanism is denoted by the imperative *m.refine*. The semantics of the two are:[21]

> *m.extend* denotes a transformation that extends the specification of *m*, i.e. $m.s' \geq m.s$
>
> *m.refine* denotes a transformation that refines the behaviour of *m*, i.e. $m.impl' \geq m.impl$

Extension and refinement of a mechanism may generate new mechanisms (decomposition). For example, when we extend the specification and refine the behaviour of method *setToNextDay* to work for varying number of days per month, it may generate the need for a method to return the number of days of the current month. And when we extend the specification and refine the behaviour of a method to deposit an amount to an account to store the transaction for later retrieval, it may generate the need for a new class to represent the concept of a transaction.

The purpose of a development process is to construct a program *P*, i.e. a set of mechanisms that fulfils their requirements. Using the terminology of section 7.1.2, we can express the goal of a development process as:

$$\textbf{goal}: \quad (\forall\ m \in P \mid : m.consistent \wedge m.total)$$

If we let *PM* denote the set of partial mechanism and *IM* the set of inconsistent mechanisms, i.e.

$$PM = \{ m \in P \mid \neg m.total \}$$
$$IM = \{ m \in P \mid \neg m.consistent \}$$

Using *PM* and *IM*, the goal can be rephrased as:

$$\textbf{goal}: \quad PM = \varnothing \ \wedge \ IM = \varnothing$$

In the following, we describe incremental development of programs (sets of mechanisms) in algorithmic form as schemas. Initially, we describe a simple development process that is gradually improved to capture more and more aspects of best practice. In Schema 7-2, we presented a sketch of stepwise improvement; a more well-defined description is provided in the following Schema 7-3.[22] [23] [24]

[21] We employ the standard notation to relate values before and after a transformation: *v* and *v'* denote the values before and after the transformation.

[22] All operations on mechanisms generalise trivially to sets of mechanisms.

[23] $\perp$ denotes the bottom value for a mechanisms implementation; it can be anything, but in practice, we provide the simplest possible implementation: an empty stub.

[24] For any relational operator, ◊, the assignment statement *v*:◊ *exp* ensures *v* ◊ *exp* (for suitable substitutions of free occurrences of *v* in *exp*).

```
$P.s, P.i := abort, \perp$ ;
**do** $PM \neq \varnothing \rightarrow m{:}{\in}\ PM$ ; $m.extend$
[]  $IM \neq \varnothing \rightarrow m{:}{\in}\ IM$ ; $m.refine$
**od**
{ **goal**: $PM = \varnothing \ \wedge \ IM = \varnothing$ }
```

Schema 7-3: *Stepwise improvement*

A typical step in incremental programming involves altering a set of mechanisms at a time. An alternative description, which more realistically captures the practice of programming, is presented in Schema 7-4.

```
$P.s, P.i := abort, \perp$ ;
**do** $PM \neq \varnothing \rightarrow M{:}{\subseteq}\ PM$ ; $M.extend$
[]  $IM \neq \varnothing \rightarrow M{:}\ {\subseteq}\ IM$ ; $M.refine$
**od**
{ **Goal**: $PM = \varnothing \ \wedge \ IM = \varnothing$ }
```

Schema 7-4: *Advanced stepwise improvement*

Refactoring is the process of changing a program in such a way that it improves its internal structure without altering its external behaviour. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, refactoring is improving the design of the code after it has been written [Fowler 1999].

If we extend our vocabulary about program transformations and the state of programs with a few new loosely defined terms[25], we can incorporate refactoring (and thereby achieve a more comprehensive description of the programming process as it is carried out by professional software developers on large scale projects) by adjusting the goal and by adding an extra guarded command to the loop. The result is shown in Schema 7-5.

```
$P.s, P.i := abort, \perp$ ;
**do** $PM \neq \varnothing \rightarrow M{:}{\subseteq}\ PM$ ; $M.extend$
[]  $IM \neq \varnothing \rightarrow M{:}\ {\subseteq}\ IM$ ; $M.refine$
[]  $BM \neq \varnothing \rightarrow M{:}\ {\subseteq}\ BM$ ; $M.refactor$
**od**
{ **Goal**: $PM = \varnothing \ \wedge \ IM = \varnothing \ \wedge \ BM = \varnothing$ }
```

Schema 7-5: *Stepwise improvement unified with refactoring*

In Schema 7-5, the first guarded command models repeated improvements of the specification, the second guarded command models refinement of the implementation to meet the current specification, and the third guarded command models refactoring. A new guarded command is added for each additional activity we may choose to include in the model.

[25] *m.wellDesigned* indicates that *m* is well designed (the opposite as “bad smell” [Fowler 1999]). $BM = \{ m \in P \mid \neg m.wellDesigned \}$. The imperative *m.refactor* is an activity that improves the design of *m*.

Software optimization is the process of modifying a mechanism to optimize its use of resources (e.g. time, space, and bandwidth) but without altering its external behaviour. Optimization can be incorporated in the process in the same way as refactoring simply by adding an extra guarded command to the inner loop.

Every other criterion to be addressed is treated in a similar way. It must be suitably incorporated in the goal and in the invariant, and it adds a new dimension to the process modeled as an extra guarded command of the loop.

This finishes our development of a model for incremental program development by stepwise improvement. In the following section we shall see how elimination of some of the non-determinism and reinterpretation of components in Schema 7-5 can lead to a model for test-driven development.

7.1.4 Programming strategies

As described in Schema 7-5, program development is navigation in an *n*-dimensional space (depending on the number of activities considered).

A *programming strategy* is a prescription of a more specific ordering and execution of the activities of the programming process.

Test-Driven Development (TDD) is an example of a programming strategy. TDD is an evolutionary approach to software development which combines the use of tests as specifications written before the production code with refactoring [Beck 2003]. Three of the more important goals of TDD are specification of required behaviour, support for refactoring, and validation of production code. The latter is the traditional purpose of testing; the first two are more innovative and relates to extreme programming and agile development methods [Beck 1999, Martin 2003].

In the context of systematic development of programs from their requirements, TDD provides a pragmatic approach to specifications. The fundamental idea in TDD is that tests are written *before* production code to serve as specification of the next chunk of functionality to be implemented. In this sense, TDD complements formal specifications of software.

Extension of the specification of a mechanism moves the boundary between the specification and the implementation; it is precisely in this situation that test cases that operationalize (the new part of) the specification must be created. Restriction of some of the non-determinism to ensure that extension is the primary driver of progression, ensuring that all m mechanisms become consistent and well-designed for each extension, and reinterpretations of the imperative *extend* (to cover development of test code) and the predicate *inconsistent*[26] (to cover execution of test code) is all that is needed to unify our description of stepwise improvement with TDD (see Schema 7-6).

[26] The predicate inconsistent is part of the definition of the first guard of the inner loop, $IM \neq \emptyset$.

```
$P.s, P.i := abort, \perp$ ;
{ **inv**: $IM = \emptyset \wedge BM = \emptyset$ }
**do** $PM \neq \emptyset \rightarrow$
    $M :\subseteq PM$ ;
    $M.extend$ ;
    { **inv**: $m \in PM \equiv m.partial \wedge m \in BM \equiv m.badSmell$ }
    **do** $IM \neq \emptyset \rightarrow M: \subseteq IM$ ; $M.refine$
    [] $BM \neq \emptyset \rightarrow M: \subseteq BM$ ; $M.refactor$
    **od**
**od**
{ **Goal**: $PM = \emptyset \wedge IM = \emptyset \wedge BM = \emptyset$ }
```

Schema 7-6: *Stepwise improvement unified with test-driven development*

Whether TDD is an effective strategy for software development is not important here. The point is that TDD seamlessly unifies with our description of an incremental development process. This, of course, increases our confidence in the viability of the description.

In TDD, as always, the initial state of mechanism $m = \langle r, s, i \rangle$ is: $s = abort$ and $i = \perp$ (partial and consistent), and we are aiming for a final state in which the mechanism is total and consistent. The potential slack between requirement and specification on the one hand $(r - s)$ and between specification and implementation on the other $(s - i)$ means that a mechanism can be total or partial and, independently, consistent or inconsistent. In a development process, all four states are legal and meaningful.

A state diagram describing development processes according to TDD (Schema 7-6) is presented in Figure 7-7 (we ignore refactoring not to clutter the diagram; > marks the initial state).

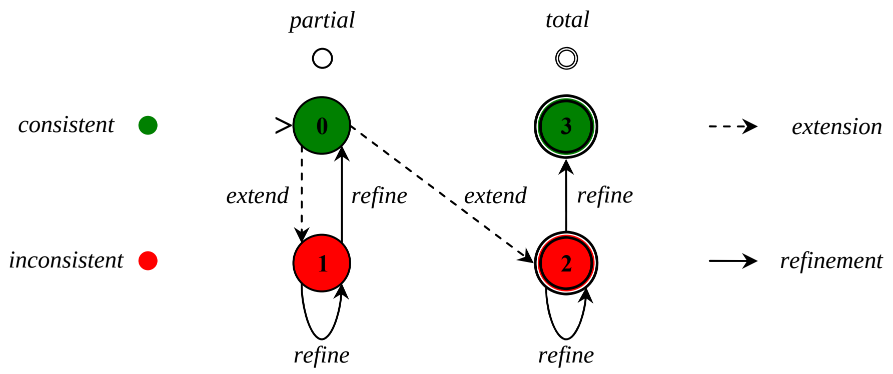


Figure 7-7: *State-transition diagram for incremental development processes*

There are degrees of partialness as well as degrees of inconsistency. This is reflected in an unlimited number of intermediate states depending on the length of the development trace (the number of intermediate specifications) and the number of refinement steps for each of the intermediate specifications. Or, to put it differently, depending on the number of repetitions of the outer loop (partialness) and inner loop (consistency) of the incremental development process described in Schema 7-4 in the previous section. A concrete development scenario is described in Figure 7-8.

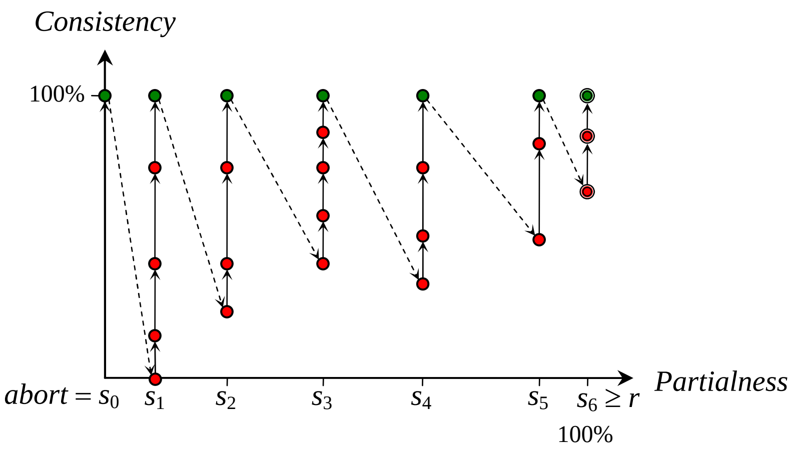


Figure 7-8: *Fine-grained states and sequence of progression*

Including quality of design as a criterion and refactoring as the activity to ensure it adds a new dimension to the state space; consequently, finding an optimal development path becomes more complicated. Similarly, as already mentioned, every other criterion to be included adds a new dimension to the state space and complicates the programmer's navigation.

7.1.5 Degrees of correctness

Traditionally, correctness is an absolute criterion; either a program is correct or it is not. In our model, correctness becomes a relative notion; furthermore, we can quantify the degree of correctness of a consistent mechanism.

Operating with two specifications, r and s, allow us to quantify the *degree of correctness* of a mechanism. Assume a consistent but partial mechanism, i.e. there is a slack between r and s, $r > s$. This means that the precondition of r is weaker than the precondition of s or that the postcondition is stronger. Another way of expressing this is that the number of initial cases that satisfies r is larger than the number of initial cases that satisfies s or that the number of results that satisfies r is smaller than the number of results that satisfies s.

Definition 14: The number of states satisfying a predicate p is denoted by $|p|$.

In general, the following properties hold for specifications.

Property 1: $(\forall\ s, t \mid s \leq t : |s.pre| \leq |t.pre| \ \wedge\ |s.post| \geq |t.post|\ )$.

Property 2: $(\forall\ s, t \mid s < t : |s.pre| < |t.pre| \ \vee\ |s.post| > |t.post|\ )$.

We can define the degree of totality of a mechanism and the degree of correctness (of a consistent mechanism) in terms of sizes of sets of states satisfying pre- and postconditions, as follows:

Definition 15: The *degree of totality* of a mechanism $m = \langle r, s, i\rangle$ with $s \leq r$ is defined as:

$$\frac{|s.pre|}{|r.pre|} \cdot \frac{|r.post|}{|s.post|}$$

Definition 16: For consistent mechanisms the degree of totality is also called the *degree of correctness*.

In the next section we provide examples of degrees of correctness of incremental versions of a mechanism.

7.1.6 Two examples

In this section, we provide two simple examples to vitalize the conceptual framework of the previous sections. We sketch the development of a mechanism representing a date and a mechanism representing an account. The paradigm is object-oriented programming, and the syntax is Java.

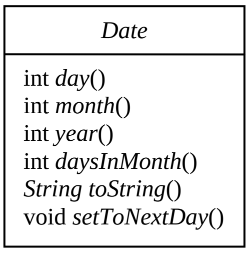
The first example, date, demonstrates how methods can be developed in a number of increments where more and more aspects of the requirements are taken into consideration.

The second example, account, demonstrates how a program with several classes can be developed in a number of increments by first focusing on a single class and a few methods of that class and later taking other classes, relations between classes, and new methods of the first class into consideration.

In both cases, we provide semi-formal specifications for the stages in the development trace. This is not to suggest that this is what programmers should do, we do it to make precise what the stages are and how they can be captured by a sequence of specifications of a monotone development trace. As we shall see in the next section, specifications can take many different forms at many different levels of abstractions.

Date

The first programming task we consider is the implementation of class *Date*, specified as follows:



Date

int *day*()
int *month*()
int *year*()
int *daysInMonth*()
String toString()
void *setToNextDay*()

Figure 7-9: *Class model for Date*

The mechanism we must develop, class *Date,* consists (at this stage) of six part-mechanisms, methods *day*, *month*, *year*, *daysInMonth*, *toString* and *setToNextDay*. The requirements of the compound mechanism *Date* is the requirements of the six part-mechanisms:

day:	r = [true, = the day of the month] [27]
month:	r = [true, = the month of the year]
year:	r = [true, = the month of the date]
daysInMonth:	r = [true, = the number of days in *month*()]
toString:	r = [true, = the date in the format dd-mm-yyyy]
setToNextDay:	r = [true, this is the next date acc. to the Gregorian calendar]

In the following we ignore the basic queries and concentrate on the derived query *toString* and the command *setToNextDay* (the query and command terminology is from [Mitchell et al. 2002]).

For each of the two methods, we suggest a monotone development trace, i.e. a sequence of specifications leading to the requirements of the method. For each specification in the development trace we denote the degree of correctness of a program with the specified behaviour.[28]

toString:	s_0: *abort*	0%
	s_1: [$day() \geq 10 \ \wedge \ month() \geq 10$, *r.post*]	8,3%
	s_2: r	100%
setToNextDay:	s_0: *abort*	0%
	s_1: [$day() < daysInMonth()$, *r.post*]	96,71%
	s_2: ["$daysInMonth() = 30$", *r.post*]	97,80%
	s_3: ["*year* is not a leap year", *r.post*]	99,93%
	s_4: ["*year* is not a century", *r.post*]	99,99%
	s_5: ["*year* is not a 4-century", *r.post*]	99,99%
	s_6: r	100%

The degrees of correctness expressed in percentage, deserves some explanation. For example, the degree of correctness of s_1 of command *setToNextDay* stems from the fact that the method misbehaves in only 12 out of 365 cases (when a *Date* object is in a state representing the last day of a month); it behaves according to the specification in the remaining 353 cases.

Remark: This example demonstrates the well-known fact that x% of the development time is devoted to 100–x% of the functionality where x is a number closer to 100 than to 0. *End of remark.*

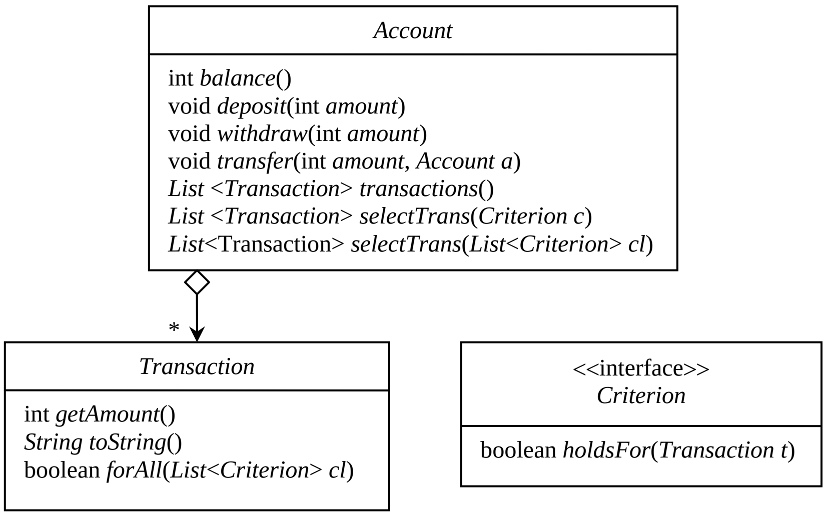
We can merge the two development traces any way we want; in total, there are ten increments in the development of the complete program, and for each increment —improvement of the specification— there is a lot of refinement to do. However, this aspect of the development process is not our concern here. A detailed description of principles and techniques to apply in the refinement process is presented in the paper in chapter 16.

Account

[27] Postconditions for queries take the form '= *exp*' indicating that the result of the query is *exp*.

[28] The preconditions in quotes are approximations to the precise (but more complicated) preconditions. For example, the precondition for *setToNextDay* in s_3 should have been $day() = 28 \wedge month() = 2 \ \Rightarrow \ year$ is not a leap year.

The next task to consider is the implementation of the following class model:



Account
int *balance*()
void *deposit*(int *amount*)
void *withdraw*(int *amount*)
void *transfer*(int *amount*, *Account a*)
List <Transaction> transactions()
List <Transaction> selectTrans(*Criterion c*)
List<Transaction> *selectTrans*(*List<Criterion> cl*)
*
Transaction
int *getAmount*()
String toString()
boolean *forAll*(*List<Criterion> cl*)
<<interface>>
Criterion
boolean *holdsFor*(*Transaction t*)

Figure 7-10: *Ultimate requirements model of Account program*

The requirements of the commands *deposit*, *withdraw*, and *transfer* are:

```
deposit:   $r$ = [true, = if $amount \geq 0 \rightarrow balance()' = balance() + amount$
                  [] $amount < 0 \rightarrow NegativeAmountException$
                  fi
           ]
withdraw:  $r$ = [true, = if $amount \leq balance()$
                     $\rightarrow balance()' = balance() - amount$
                  [] $amount > balance \rightarrow NoCoverageException$
                  fi
           ]
transfer:  $r$ = [true, = if $0 \leq amount \leq balance()$
                     $\rightarrow balance()' = balance() - amount \ \wedge$
                        $a.balance()' = a.balance() + amount$
                  [] $amount > balance \rightarrow NoCoverageException$
                  [] $amount < 0 \rightarrow NegativeAmountException$
                  fi
           ]
```

Each of the commands *deposit*, *withdraw*, and *transfer* generates a *transaction*, which can later be retrieved by suitable queries. The functional requirements of the queries *transactions*, *selectTrans* (both of them), and *forAll* are as follows:

transactions:	r = [true, = [t | t corresponds to a command on this]]
selectTrans(*c*):	r = [true, = [$t \in transactions$ | $c.holdsFor(t)$]] [29]
selectTrans(*cl*):	r = [true, = [$t \in transactions$ | $forAll(cl)$]]
forAll(*cl*):	r = [true, = (($\forall\ c \in cl$ |: $c.holdsFor$(this)))]

[29] [$e \in aList \mid p(e) : f(e)$] is a list builder expression that describes the list of elements $f(e)$ where e is the elements from *aList* that satisfy predicate p. If f is the identity function, the last part can be omitted and the expression written as [$e \in aList \mid p(e)$].

To demonstrate that new requirements may be added during the development process, assume that the requirements are provided in two stages. The initial requirements are captured by the model in Figure 7-11, and the ultimate requirements are those presented in the model in Figure 7-10.

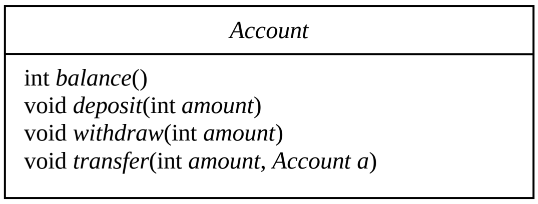

Account
int *balance*()
void *deposit*(int *amount*)
void *withdraw*(int *amount*)
void *transfer*(int *amount*, *Account a*)

Figure 7-11: *Initial requirements model of Account program*

Stage 1: In this stage, no requirements relate to transactions; thus, the three commands *deposit*, *withdraw*, and *transfer* are not requested to generate transaction objects. For each of the three commands, an obvious development trace presents itself:

s_0: *abort* s_1: normal case s_2: special case (exception)

This gives in total six increments in the process for this stage (plus the first step of producing stub methods to implement *abort*).

Stage 2: In this stage we address the "new" requirements which, as always, can be broken down into a number of increments.

Applying a bottom-up or do-simple-things-first strategy, a natural first thing to do is to implement class *Transaction* while ignoring its relation to the rest of the system including method *forAll*.

The next obvious thing is to implement the query *transactions* and modify commands *deposit*, *withdraw*, and *transfer* to generate a *Transaction* object for each normal invocation (and add it to a suitable collection object).

The last thing to do is to implement the small set of framework methods (*forAll*, *selectTrans*(*c*), and *selectTrans*(*cl*)) that allow general selection criteria to be created in clients of class *Account*.

In doing so, the programmer may adopt an opportunistic or depth-first approach and go to a more detailed level to get enough information to understand the task to be performed, as discussed in section 6.2.3. The programmer may want or need to make experiments in order to gain understanding of the intended behaviour and potential implementation of the generic framework methods. One way to do this could be to invent a new selection method with a hard-wired selection criterion:

$r = [\text{true}, [\ t \in transactions \mid lower \le t.getAmount() \le upper\ ]\ ]$
List<*Transaction*> *selectTransInInterval*(int *lower*, int *upper*)

Once this method is implemented, the programmer may have gained enough information and insight into the problem to generalise the specific solution of method *selectTransInInterval* to the generic method *selectTrans*(*c*), and from this to method *selectTrans*(*cl*).

In total, stage 2 as described here encompasses five to ten increments (depending on the granularity), and for each increment a number of refinements. As in the first example, a systematic incremental process is crucial for controlling the complexity of the task.

As indicated by these two simple examples, a systematic, incremental programming process reduces the total programming task to a number of relatively simple and in some cases even trivial subtasks that can addressed and verified one at a time. Clearly, we need to teach novices about the programming process, and we need to initially provide strong guidance about which increments to make and in which order to make them. Without process guidance, novices will struggle with many different aspects of a task at a time and achieve little. The inevitable result is a chaotic, self-made programming process and loads of frustration, which the novice is most likely to interpret as either frustration and anger at the instructor for posing such hard assignments or personal incompetence. Incompetence it is indeed —but on the side of the educator who miserably fails to properly educate novices in the skills of programming.

7.2 Incremental development and OOP

In the previous section, we presented a conceptual framework for program extension and characterized an incremental programming process independently of a particular paradigm, stepwise improvement. In the last subsection, we gave two simple examples of applications of the theory in the context of object-oriented programming. In this section, we describe the nature and role of specifications at various levels of abstraction in object-oriented programming. In classical programming methodology, the loop body and the loop invariant is developed hand in hand with the latter leading the way [Gries 1981, p. 164]. In the context of object-oriented programming we broaden the perspective and argue that implementation and specification is developed hand in hand with the latter leading the way.

7.2.1 Programming as a modeling process

In [Madsen et al. 1993, p. 284], the object-oriented perspective on programming is defined as follows: "A program execution is regarded as a physical model simulating the behaviour of either a real or imaginary part of the world." The real or imaginary part of the world being modeled is called the referent system, and the program execution constituting the physical model is called the model system.

The programming process (initiated by a vision of a new system) involves identification of relevant concepts and phenomena in the referent system and representation of these concepts and phenomena in the model system. This process consists of three subprocesses: abstraction in the referent system, abstraction in the model system, and modeling (no particular ordering is imposed among the subprocesses).

Fowler distinguishes between three abstraction levels for interpretation of class models: conceptual level, specification level, and code/implementation level [Fowler et al. 2000]. Fowler's levels of interpretation correspond to the results of the three subprocesses mentioned above: a conceptual model (expressed in terms of problem specific concepts), a specification model (expressed in terms of realised concepts, and an implementation model (expressed in terms of objects). Figure 7-12 illustrates the programming process as a modeling process between a referent system and a model system.

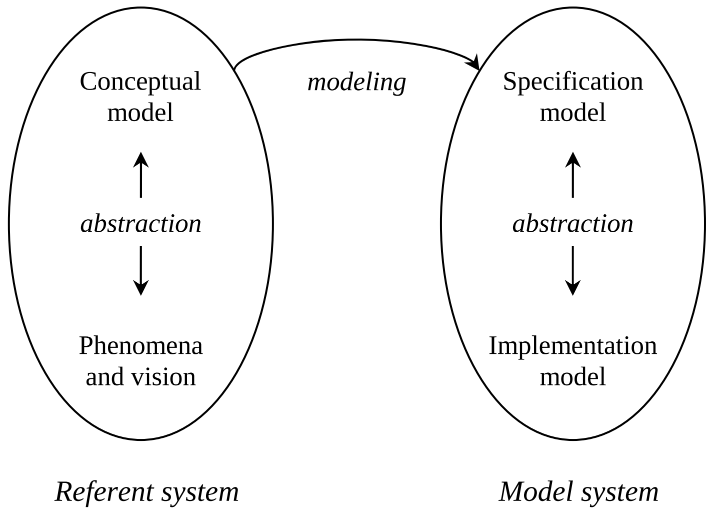


Figure 7-12: *Programming as a modeling process*

Specifications play a key role in the incremental programming processes described in the previous section. In the context of object-oriented programming, we shall assume that the specification model is expressed as a model with relevant sub-models (a static class model, a dynamic state model, functional specifications of methods, etc.). When constructing specification models, all relevant notations and tools may be used, e.g. UML with class diagrams, state charts, and the object constraint language, Petri nets, predicate logic, and test suites.

As indicated in section 6.3.3, we adopt a non-standard, horizontal approach to programming education in which novices are provided with worked examples and initially do very simple tasks and then gradually do more and more complex tasks, including design-in-the-small by adding new classes and methods to an already existing design. Our approach is in concordance with Buck and Stucki. In [Buck et al. 2000], the authors argue that "traditional approaches to CS1 and CS2 are not in congruence with cognitive learning theory" and provide arguments for a reversed order of topics based on Bloom's classification of educational objectives [Bloom et al. 1956]. The title of Buck and Stucki's paper is "Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development", and the message of the paper is that the ordering of topics that best matches Bloom's hierarchy of cognitive development is the reverse of the order of activities in the classical software lifecycle model. The students first do implementation of methods within an existing design; later they move to design; and analysis and requirements are covered in later courses.

In our current course design, we more or less ignore two of the subprocesses described in Figure 7-12 and restrict ourselves to the task of implementing and expanding specification models expressed as class diagrams, informal functional specifications of methods, and test suites.[30]

7.2.2 Implementing specification models

When implementing specification models, we identify three independent activities: (1) implementation of inter-class structures, i.e. relations between classes and methods that maintain these relations; (2) implementation of intra-class structures, i.e. the internal structure and representation of a class; and (3) implementation of methods. Of course, (3) is logically part of (2), but because different principles and techniques are involved, we prefer to separate the two.

The principles and techniques that relate to the three activities are: (1) standard coding patterns for the implementation of relations between classes; (2) class invariants and techniques for evaluating these; and (3) algorithm patterns (e.g. sweep, search, divide-and-conquer) and loop invariant.

The loop invariant is a prime software engineering tool that allows understanding each independent part of the loop —initialization, termination, condition, progressing toward termination— without having to look at the other parts [Gries 2006]. Similarly, the class invariant is a prime software engineering tool that allows us to separate consideration and evaluation of alternative representations from implementation of the methods of the class and to implement each method without worrying about the others. And the same story goes for class modeling, which is a prime software engineering tool that allows us to separate specification of each class from specification of the relationship between classes. The fundamental and recurring principle of separation of concerns permeates the techniques of all three activities. Figure 7-13 summarize activities and associated programming techniques for implementation of specification models.

Activity	**Techniques**	**Characteristics**
Implementation of inter-class structure.	Standard coding patterns for the implementation of relations between classes (aggregation and associations, both with varying multiplicities).	Separation of the specification of each class from specification of relationships between classes. Standard implementations of relation types (supports schema creation and transfer).
Implementation of intra-class structure.	Class invariants and techniques for evaluating these.	Separation of consideration and evaluation of alternative representations from implementation of the methods of a class. Separate implementation of each method without worrying about the others.
Implementation of methods.	Algorithmic patterns and loop invariants.	Standard implementation of algorithm patterns (supports schema creation and trans-

[30] As mentioned, the students do design-in-the-small, but the major emphasis is on implementing designs provided by the teacher.

		fer). Separation of initialization, termination, condition, and progression.

Figure 7-13: *Activities and associated programming techniques for implementation of specification models*

In chapter 9, we present the instructional design of an introductory programming course where an incremental programming process and practise of these activities and techniques constitutes the primary contents of the course. Increasingly complex specification models define the course progression, not constructs of the programming language, as is the custom. Teaching aspects of implementing inter-class structures are described in section 9.2.3 and 9.2.4; the aspect of intra-class structure is covered in section 8.2 (as part of our novice's process STREAM) and 9.2.5; and method implementation is covered in section 8.2.6, 8.2.7 and 9.2.9.

7.3 Conclusion

This chapter has addressed two research questions

$Q_{4.3}$: How does best-practice in modern software development relate to programming methodology?

$Q_{4.4}$: Can we provide a characterization of the programming process that unifies programming methodology and best-practice of modern software development?

With specifications as the common denominator, we have presented a conceptual framework for program extension that captures the essence of incremental program development and unifies classical stepwise refinement with best practices of software development. In our model, stepwise refinement is a sub-process of incremental development.

Our framework enables us to define correctness as a relative notion and to quantify the degree of correctness of a mechanism.

Two simple examples were used to demonstrate how to put the conceptual framework into practice and to argue (once again) for the necessity of teaching novices about the programming process and to provide guidance about which increments to make and in which order to make them.

8 A Programming Method for Novices

It is time to address the most important of our five research questions

> **Research question 5** (Q_5): How can we educate novices in the skills of programming? The question is refined to four more specific questions:
>
> $Q_{5.1}$: How can we scale down modern software development methods to a programming process for novices?
>
> $Q_{5.2}$: How can we structure the relevant body of knowledge so that it can be most readily grasped by the learner?
>
> $Q_{5.3}$: How can we organize efficient learning paths/courses that incrementally approximate best-practice in modern software development at the level of novices?
>
> $Q_{5.4}$: How can we adopt results of cognitive science and educational psychology to the instructional design[31] of introductory programming education?

In this chapter, we address $Q_{5.1}$ and $Q_{5.2}$. The two remaining questions are addressed in chapter 9.

In section 8.1, we criticise traditional approaches to problem-solving methods and present an alternative approach for providing guidance to students regarding the programming process. Section 8.2 provides a brief presentation of STREAM — a scaled-down programming process for novices shaped by the model for incremental program development of the previous chapter. The presentation of the process is structured so that it can most readily be grasped, remembered, and applied by the learner. In section 8.3, we illustrate application of STREAM using the recurring Date example. Section 8.5 is the conclusion of the chapter.

The work reported in section 8.1.3 was carried out with Bennedsen and is described in detail in the paper in chapter 15. The work reported in section 8.2 was carried out with Kölling and is described in detail in the paper in chapter 16.

8.1 Random walk or guided tour

In the perspective of chapter 7: incremental development through stepwise improvement, programming is about finding one's way through the lattice of specifications $\langle abort, req \rangle$ spanned by the requirements of the program to be

[31] Instructional design concerns detailed specification of teaching/instruction as opposed to curriculum design which concerns specifications of learning outcome.

built. For each specification in the development trace, a number of refinements are to be made in order to reach a consistent program according to the current specification. And for each refinement, there are concerns such as refactoring, test case development, optimization, etc. to deal with. For novices, refactoring and optimization may be ignored, and test cases can be provided; however, programming style is an additional concern that must be addressed (indentation, naming, documentation, etc.); thus, a 3-dimensional space of extensions, refinements, and styling must be navigated (see Figure 8-1). For a novice, this is a daunting task with immense sources of cognitive load!

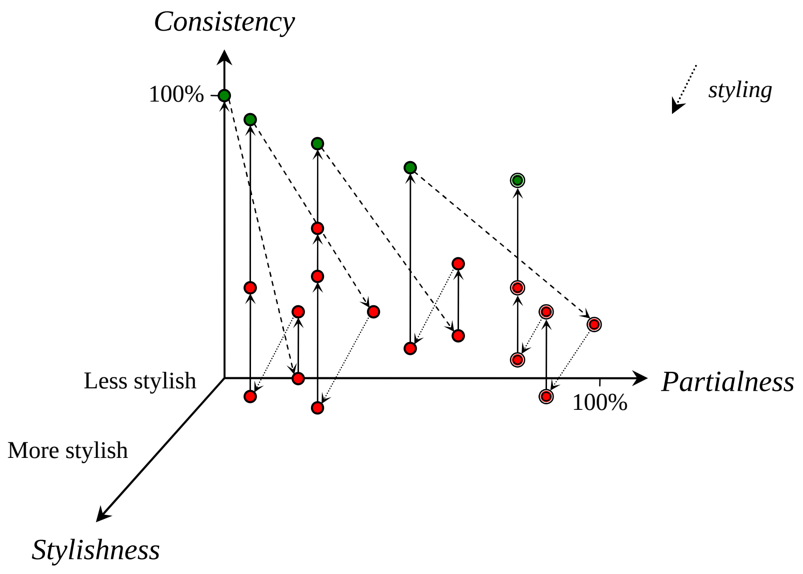


Figure 8-1: *The 3D maze of partialness, consistency, and stylishness*

8.1.1 Random walks

The traditional approach to programming education is to "invite" the students for a random walk in the 3D maze. Students are shown a few finished programs and told to solve programming problems on their own. "The more, the better", the saying is. However, as in the case of the experiment of section 3.2, which demonstrated that no learning takes place if the cognitive load of the problem to be solved is too high, we may not expect much learning to take place from that style of programming education. It is not surprising that many students give up and never learn to program.

Actually, it is even worse than indicated above. As mentioned in chapter 7, monotone development traces are rare, and for novices, they are unlikely to occur at all since novices are uncertain about what they are aiming for. In conclusion, there are all sorts of traps and pitfalls in the maze, and it is not surprising that many get lost.

Attempts have been made to provide guidance by describing how to solve problems. In *How to Solve It* [Polya 1957], the author describes techniques for mathematical problem solving based upon the following four-step plan:

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan.

4. Look back and check the result.

Of course, the four-step process is very reasonable, but not very constructive.

Hyman and Anderson have discussed problem-solving in a wider context [Hyman et al. 1965]. Naur has pointed to the work of Hyman and Anderson on several occasions in 1970, 1972, and 1981 [Naur 1972, Naur 1992a, Naur 1992b]. Hyman and Anderson's discussion is centered on eight rules for problem solvers:

1. Run over the elements of the problem in rapid succession several times, until a pattern emerges that encompasses all these elements simultaneously.
2. Suspend judgement. Don't jump to conclusions.
3. Explore the environment. Vary the temporal and spatial arrangement of the materials.
4. Produce a second solution after the first.
5. Critically evaluate your own ideas. Constructively evaluate those of others.
6. When stuck, change your representational system. If a concrete representation isn't working, try an abstract one, and vice versa.
7. When stuck, take a break.
8. Talk about your problem with someone.

Again, these are reasonable rules, but not very constructive.

A third classic in problem solving is Rubinstein's *Patterns of Problem Solving* [Rubinstein 1975]. Rubinstein recommends that the following tasks be part of the problem solving process:

1. Write down the problem in its primitive form.
2. Transform it to simple language.
3. Translate it to mathematical statement, if possible.
4. Represent using diagrams, charts, and graphs.

In hindsight, the rules and recommendations of Polya, Hyman and Anderson, and Rubinstein may make sense, but they provide no help whatsoever for a novice addressing a concrete programming problem.

Recently, Deek has proposed *The Software Process: A Parallel Approach through Problem Solving and Program Development*. Deek writes, "The development of this process was based on an extensive study of problem solving methodologies" [Deek 1999]. Deek's process, which is a blend of Polya's and Rubinstein's problem solving methods, consists of six steps:

1. Formulate the problem.
2. Plan the solution.
3. Design the solution.
4. Translate the solution (into executable form).
5. Test the solution.
6. Deliver the solution.

Besides being merely a rephrasing of Polya's method for mathematical problem solving, Deek's process is nothing but a scaled-down version of the now abandoned waterfall methods of software engineering, which carry strong reminiscences of program development through stepwise refinement.

Over the years, others have proposed programming methods based on Polya's methods for problem solving, e.g. [Perkins 1981, Proulx et al. 2006, Raadt et al. 2004b, Tu et al. 1990].

In spite of good intentions, novices are provided little help from these high-level recommendations of common sense behaviour.

In our endeavour of providing guidance, we need to be much more specific about principles and techniques of (incremental) program development. We know this from loads of research from the past 25 years. As mentioned in our closing paragraph of section 4.2.1 (*Psychological studies*), study after study, even multi-institutional and multinational, have thoroughly documented that students cannot program and that the major problems they experience are composition-based —how to put the pieces together. We have a long-standing problem of international scale, which we are aware of, and yet we persist to teach programming primarily by explaining language constructs and show-casing finished programs —even though it is procedural knowledge and strategies for putting the pieces together that are needed!

8.1.2 Guided tours

Our approach to programming education, which draws upon the results of cognitive load theory, offers an alternative to endless random walks. Instead, we suggest guided tours. By providing guidance and scaffolding[32] with respect to all dimensions involved, we can ensure that students exercise the important aspects of programming while keeping the cognitive load within the bounds where learning outcome is optimized.

As mentioned in section 3.1.2, research on expert-novice differences in problem solving and cognitive skill acquisition indicate that speed and accuracy of experts is not accomplished by major, qualitative changes in their problem solving strategies [VanLehn 1989, p. 563]. The effects of their expertise are more subtle. For instance, whenever an expert and a novice are deciding which chess move to make, both consider the same number of moves and investigate each move for about the same amount of time. The difference is that the expert considers only the good moves and usually chooses the best one, whereas the novice considers mediocre moves as well, and often does not choose the best move from those considered. Thus, expertise lies not in having a more powerful overall strategy or approach but rather in having better knowledge for making decisions at the points where the overall strategy calls for a problem-specific choice. Similarly, experts seem better at monitoring the progress of their problem solving and allocating their efforts appropriately. [Schoenfeld 1981] concludes that *metacognitive or managerial skills* are of paramount importance in human problem solving. The same sort of managerial monitoring is evident in numerous

[32] Scaffolding is a term from cognitive apprenticeship describing support provided by the master to apprentices in order to carry out some given task: "this can range from doing almost the entire task for them to giving them occasional hints on what to do next" [Collins et al. 1991, p. 7].

studies including studies of programmers and software design [Jeffries et al. 1981].

Our primary means of providing guidance with respect to incremental development is through the structure of the teaching material (textbook, exercises and assignments, and videos) and an apprentice-based teaching approach. Guidance with respect to refinement is provided through a carefully designed novice's process of object-oriented programming. The process, which we call STREAM, is described in the next section.

The textbook we employ [Barnes et al. 2006] is problem-based and written according to the pedagogical principle of apprentice-based programming education (see *Case studies and apprenticeship* in section 4.2.3). In [Kölling et al. 2004], the authors describe in detail the rationale, motivation, and goals of the approach of the book. Following Barnes and Kölling's approach, first students *observe* the teacher demonstrating and extending an existing piece of software using new techniques or constructs introduced for the purpose, then students *apply* the new material to the project under guidance, and finally the students *design* their own tasks as extensions of the project at hand. In this way, the order of student activities is exactly reversed compared to classical, clean-slate assignments; there, students typically have to start with design, followed by applying new material before they observe behaviour.

The exercises and assignments we employ are carefully written to guide the student through a specific path of program development. Beacons of the development trace are provided, and so is when to make turns in the maze, i.e. when to extend, when to write specifications (Javadoc), when to write test code, when to refine, when to refactor, and when to style the code.

Example: This is an example of a closed-lab exercise provided to the students in the fourth week of the introductory programming course. For comparison, we provide a programming exercise that corresponds to the development presented in section 7.1.6 (Date).

Consider the following class diagram:

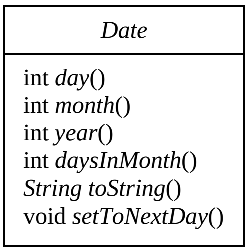


Figure 8-2: *Class model for Date*

The requirements of the methods of class *Date* are:

day:	= the day of the month
month:	= the month of the year
year:	= the month of the date
daysInMonth:	= the number of days in the month
toString:	= a presentation of the date in the format dd-mm-yyyy
setToNextDay:	set the date to the next date acc. to the Gregorian calendar

1. Build a walking skeleton, i.e. a class *Date* with stubs for each of the six methods.
2. Write a program to test the behaviour of well-chosen Date objects.
3. Consider the following two alternative representations for class Date. R_1: three integer variables representing day, month, and year of the date. R_2: one integer variable representing the number of days since January 1, 1970. Make a *REM* for these two representations and fill it out using the *TEACH* scale.[33]
4. Choose the representation that makes your job of implementing the class as easy as possible.

The following part of the exercise assumes that the students have chosen representation R_1. The exercise has an alternative second part tha provides guidance for the implementation of R_2.

5. Declare field variables for the chosen representation and provide at least one constructor for the class. Test your solution.
6. Implement methods *day*, *month*, *year*, *daysInMonth*, and *toString*. You may assume that there are 30 days in every month. Test your solution.
7. Make an implementation of *setToNextDay* that works provided the date is not the last date of the month (i.e. day $\leq$ 30). Test your solution.
8. Improve your implementation of *setToNextDay* so it works also in the special case where the day is the last day of the month. Make sure you use method *daysInMonth* whenever you need to refer to the number of days in the current month. You may ignore the case where the date is the last day of the year (December 30 according to the current assumptions). Test your solution.
9. Improve your solution of *setToNextDay* so it also works for the special case of the last day of the year. Test your solution.
10. Improve your implementation of *daysInMonth* to return the correct number of days of the current month. At this point, you may ignore the special case of leap years. Test your solution.
11. Improve your implementation of *daysInMonth* to work correctly also in the special case of leap years. Hint: Make a helper method *leapYear* that returns a boolean indicating whether the current year is a leap year or not. Test your solution.

The first five steps of the assignments correspond to the first five of six steps in the STREAM method; the remaining steps of the assignment correspond to applications of specific rules that are concrete incarnations of the so-called *Mañana Principle* related to the sixth step of STREAM (our version of stepwise refinement). *End of example*.

The amount of scaffolding and guidance in this example is carefully adjusted to the average level of the students at the time of giving the exercise. In general, a programming task can be used at one of many different times during a course as long as the amount and nature of scaffolding and guidance is adjusted to the level of the (average) students.

The next example demonstrates how much we can fade our guidance over a period of three weeks (from week four to week seven); the example is one of the assignments from the final exam of the introductory course, which lasts seven weeks. The exam is described in detail in section 9.2.8 and in the paper in chapter 21.

[33] REM and TEACH refer to well-defined concepts in the STREAM programming process presented in section 8.2.

Example: Make a Java program that implements the following class model including a test program that demonstrates that your solution is correct according to the specification provided below.

The program administers customers of a company and keeps track of the amount that each customer has spent at the company.

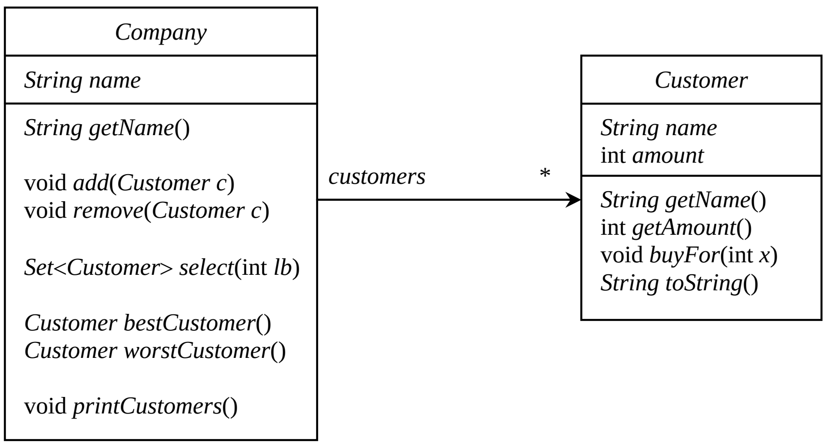


Figure 8-3: *Class model for final exam assignment*

Classes *Customer* and *Company* have the following requirements:

Person	*getName*:	= the name of the person
	getAmount:	= the amount the person has bought for
	buyFor:	increases the amount bought for by *x*; the method is called to simulate a bargain
	toString:	= a string representation of the person
Company	*getName*:	= the name of the company
	add(*c*):	add *c* to the set of customers
	remove(*c*):	remove *c* from the set of customers
	select(*lb*):	= the set of customers who have bought for an amount of at least *lb*
	bestCustomer:	= a customer who have bought for most
	worstCustomer:	= a customer who have bought for least
	printCustomers:	print the customers one per line sorted alphabetically by last name

Hint: The interfaces *Comparable* and *Comparator* (and associated methods in class *Collections*) may be exploited for implementation of the methods *bestCustomer*, *worstCustomer*, and *printCustomer*. *End of example*.

The scaffolding techniques described in this section is used primarily in order to provide guidance in the complex activity of program development. However, scaffolding does more than that and Rosson and Carrol points out: “We add to these pedagogical constructs the simple but powerful notion that if the scaffolding mirrors the structure of a complex task, it not only makes the task attainable, but also conveys a method for accomplishing similar tasks in the future” [Rosson et al. 1996].

8.1.3 Cognitive apprenticeship using videos

Providing a method of programming, detailed guidelines, and scaffolding are important aspects of educating novices in the skills of programming. However, a *description* of intended programmer behaviour is only part of the story; we also need to *demonstrate* how to apply programming methods, techniques, and guidelines in practice. An apprentice-based approach to teaching involves demonstrating the master's programming process to the apprentice.

The theory of cognitive apprenticeship holds that masters of a skill often fail to take into account the implicit processes involved in carrying out complex skills when they are teaching novices. To combat these tendencies, cognitive apprenticeship is designed, among other things, to bring these tacit processes into the open, where students can observe, enact, and practice them with help from the teacher [Collins et al. 1989, Collins et al. 1991].

Live programming in class is one way of doing this, but the implicit assumption is that everybody in the class is synchronized with respect to the level and pace of the presentation, which of course they are not. Some students will be more advanced and some will be less advanced. The more advanced students will waste their time because they have already grasped the principles while other students quickly will fall behind and get lost because they cannot follow the pace.

A more adaptable approach is to prepare videos that show program development sessions where the master demonstrates application of the programming methods, principles, and techniques that are emphasised in the course. (A video is a narrated screen capture of a programming session.) Apart from demonstrating major issues such as methods, principles, and techniques, a video may also be used to reveal minor details of experts' programming process —details that are so common to us that we fail to teach them explicitly to the students. The approach is more adaptable because the more advanced students may just view the video once (maybe winding through parts of it or skipping it altogether) while the less advanced students may pause, rewind, and review the video to fully understand the material.

Some of the activities that can be revealed through videos are incremental development, model-driven development, test-driven development, refinement, refactoring, compiler error handling, debugging, use of online documentation, and use of the IDE. The paper in chapter 15 provides a more complete discussion of ways of revealing the programming process to novices.

8.2 STREAM

As mentioned in the previous section, guidance with respect to refinement is provided through a carefully designed novice's process of object-oriented programming. The process, which we call STREAM, is described and illustrated in this section. STREAM is an acronym of the six activities constituting the process: *stubs*, *tests*, *representation*, *evaluation*, *attributes*, and *methods*.

Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming that appear to deal with ‘program’ as a noun rather than as a verb. We present a set of principles and techniques as well as an informal but systematic process of decomposing a programming problem. A by-now-well-known example is used to demonstrate the application of process and techniques. The process is a carefully scaled-down version of a full and rich software engineering process particularly suited for novices learning object-oriented programming. In using it, we achieve two things: to help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the aspects of software engineering.

Our techniques do not address the analysis phase or the finding of the classes from the problem domain. This may be achieved by using the noun/verb method or other simple methodologies. More likely, in very early student exercises, the teacher or the textbook will provide the class structure

In the sub-sections 8.2.1-8.2.6, we describe, in a general way, the STREAM method, which is a sequence of six simple steps that can be followed to implement classes whose intended behaviour is essentially understood. These sub-sections are kept brief and are intended as an initial overview. In sub-section 8.2.7, we present the mañana principle, which is a general principle that supports decomposition and refinement through separation of specification and implementation.

8.2.1 Stubs

We assume that the classes and their observable (public) functionality is understood and given, for example in the form of a Java interface or carefully written Javadoc comments. The first step toward implementation is to create an implementation class that implements this interface (or, if the interface is not formally given, provides methods with the intended signatures). The method implementations at this stage are stubs (i.e. minimal method bodies), and the class constitutes what Cockburn calls “a walking skeleton” [Cockburn 2002]. This is the practical interpretation of the bottom value of mechanisms from the previous chapter ($\perp$). We repeat this for every class in the project.

8.2.2 Tests

Once method stubs have been defined, test cases can be written for every method. This is commonly done using JUnit [Object Mentor Inc. 2006]. Several educational tools support JUnit testing (e.g. BlueJ and Dr. Java [Allen et al. 2002, Kölling et al. 2003]), and in environments that support recording of interactive testing, such as BlueJ [Kölling 2003a], the existence of stubs enables the test interaction to be recorded.

Tests can be provided by the teacher, the students can be requested to write tests themselves, or a combination of the two. The important thing is that a test suite exists because it acts as specification.

8.2.3 Representations

The next step aims at deciding on an implementation representation for the objects to be defined. The representation is defined by the attributes of the class. For every class, alternative representations must be considered. These can be as many as a student can think of (or the teacher provides for the student to think of), but must be *at least two*. We label each of our candidate representations R_1 to R_n.

8.2.4 Evaluation

Next, we create a *Representation Evaluation Matrix* (REM). A REM is a table with one column for each candidate representation and one row for each method in the class to be implemented (Table 8-1). We use this matrix to compare each method that must be implemented for each possible representation invariant. The comparison criteria may vary —leading to different tables— but initially it is always “implementation effort”. Above the table is a short description of each representation alternative.

R_1: *A short description of the first representation alternative.*
R_2: *A short description of the second representation alternative.*

IMPLEMENTATION EFFORT	R_1	R_2
***method*₁()** →		

Table 8-1: *Implementation effort evaluation matrix*

Table 8-1 shows an example of an *Effort REM*. In this table, we compare the estimated effort it takes to implement each method using a particular representation invariant. As values, we use a small ordered set of effort qualifiers: *Trivial*, *Easy*, *Average*, *Challenging*, and *Hard* (the “*TEACH* scale”). The qualifiers are subjective, but that is the whole point; the programmer must evaluate implementation effort according to their level of competence.

Later in the course, different REMs may be used for other criteria that are explicitly mentioned in the task specification. For example, if runtime performance is an explicitly stated goal, a *Performance REM* expressed in terms of big O may be used.

It is crucial not to judge representations on imaginary requirements. Especially, performance consideration should *not* play a role in early exercises, and it should be made clear that performance is entirely irrelevant for judgement of the Effort REM. We recommend focusing on Effort REMs in early exercises.

Initially the instructor can supply the REM, but gradually the students should be responsible for filling in the REM.

Once the Effort REM is complete, we choose the representation that is judged to have the simplest overall implementation effort.

8.2.5 Attributes

When we have settled on one particular representation, we can refine our implementation class. We now define the fields needed to represent objects. The field definitions may include their role (in the form of a comment) and possible constraints on their values (also in comment form). At this stage, we also provide appropriate initialisations for the fields, either in the form of default values or by using client-supplied values. This includes at least partial implementation of the class's constructor(s).

8.2.6 Methods

This step is actually more than a single step: it has the form of a nested loop and is derived directly from Schema 7-6.[34] The definition is:

```
do not all methods are total $\rightarrow$
    select method $m$ where $\neg$ $m.total$ ;
    $m.extend$ ;
    do $\neg$ $m.consistent \rightarrow m.refine$  od ;
od
```

Schema 8-1: *Incremental implementation of methods*

The imperative *m.extend* represents creation or retrieval of suitable test cases, and the predicate *m.consistent* represents the activity of running the tests.

The order in which methods are chosen is essentially arbitrary. Our recommendation for students who are not entirely confident is to implement the methods in order of increasing implementation effort according to the Effort REM. Of course, if some methods can be implemented using calls on others, the others should be implemented first.

It is easy to see that this completes the implementation. If the programmer successfully completes this step, the class is finished.

All the magic now lies in the "Implement method" step. This is still a large task, and it needs further advice to break it down into smaller steps.

8.2.7 The mañana principle

Implementing a method is potentially a large and non-trivial task. We aim to provide a process that breaks this task into smaller steps as well. This time, we cannot give a single recipe, since details of the method may vary widely. Instead, we give a set of rules that can be applied in certain cases.

Some methods, of course, consist of only a few lines of code and may be easy to write. Our rules aim at breaking all methods down into smaller chunks, until they approach the complexity of those easy-to-write methods. This is essentially a small variation of stepwise refinement [Wirth 1971].

[34] For consistency, we maintain terminology and notation from chapter 7.

At the heart of this technique is the *Mañana Principle,* which says:

When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later.[35]

Thus, the Mañana Principle encourages separation of concerns and the use of many small methods. We discuss an example below.

To get beginners used to the Mañana Principle, there are some more specific forms of this rule, each of which states a more concrete situation in which this principle should be used. They are:

Special Case rule: If you write code to treat a special case in your algorithm, treat the special case in a separate method.

Nested Loop rule: If you have a nested loop, move the inner loop into a separate method.

Code Duplication rule: To prevent writing the same code segment twice, move the segment into a separate method.

Hard Problem rule: If you need the answer to a problem that you cannot immediately solve, make it a separate method.

Heavy Functionality rule: Prevent a sequence of statements or an expression from becoming long or complicated by moving some of it into a separate method.

The helper methods created as a result of these rules are usually private, unless they are created in different classes. We discuss this further below.

It is important to remind students that these separate methods do not need to be implemented straight away. The calling method can be written as if the method existed. Following this, a stub for the Mañana method should be created. (If the programming environment had specific tool support for the Mañana principle, this could be automated by the IDE.)

The specific rules are initially easier to apply, because they provide concrete hints to times when they should be applied. They are, however, just instances of the Mañana Principle, and, if applied regularly, develop a coding habit that encourages the understanding and application of the principle in general.

This principle —and the derived rules— may sound abstract or complicated when presented in this theoretical form, but they are quite easy to understand when presented in the context of an example. In the next section, we discuss the development of the recurring class *Date* to illustrate these techniques in practice.

[35] The word *mañana* means *tomorrow* in Spanish. A more popular version of the principle is: don't do today what you can postpone until tomorrow. It is a very concrete form of separation of concerns.

8.3 An example

We demonstrate the techniques discussed above in the context of a simple programming problem: the implementation of the familiar class *Date*.

Here, we give the specification of the problem as a Java interface with Javadoc. It could be presented more informally as well as more formally; the introduction of interfaces is not a requirement for this process.

```
interface IDate {
  /**
   * Advance the date to the next day
   */
  void setToNextDay();

  /**
   * = a string representation of this date
   * in the format yyyy-mm-dd
   */
  String toString();
}
```

Figure 8-4: *Specification of Date*

8.3.1 Stubs

The first step is to create a class for the implementation that contains method stubs. The resulting class is presented in Figure 8-5. When the specification is provided in the form of a Java interface, this step is essentially mechanical and could be automated by a development environment. For students in early stages of learning, however, it might help to write this class skeleton by hand. The important thing is: simple rules can be given to guide the creation of this class.

```
/** An instance contains a date */
class Date {
  /**
   * Advance the date to the next day
   */
  public void setToNextDay() {
  }

  /**
   * = a string representation of this date
   * in the format yyyy-mm-dd
   */
  public String toString() {
    return null;
  }
}
```

Figure 8-5: *Date class with method stubs*

8.3.2 Tests

The next step is to ensure that appropriate test cases exist. Our techniques do not necessarily prescribe a strict test-first approach, in which students create tests for all methods themselves. A viable alternative for early programming tasks is to use teacher-provided tests. The teacher may provide a test suite for the expected methods as part of the specification of the task. The important step here is to ensure that tests exist, can be compiled, and can be executed (but do not need to pass).

We do not present the specific tests, since the actual test development is not the main focus.

8.3.3 Representations

The next step in our technique is to consider alternative representations (at least two). An obvious representation for this problem is to use three variables: day, month and year; we denote this alternative by R_1. An alternative representation is the number of days from a certain start date; we denote this alternative by R_2.

8.3.4 Evaluation

R_1 simplifies the implementation of *toString*, whereas the implementation of *setToNextDay* will be more challenging, since it must deal with the special case of the last day of a month. R_2 leads to a simple implementation of *setToNextDay* (a simple increment), whereas implementing *toString* will be hard. The result of this analysis is the Effort REM for Date (Table 8-2).

R_1: Three variables day, month, and year.
R_2: One variable counting the number of days from an origin.

IMPLEMENTATION EFFORT	R_1	R_2
***setToNextDay*()**	Challenging	Trivial
***toString*()**	Trivial	Hard

Table 8-2: *Estimate of required effort to implement Date*

We choose to use R_1 for our class, since it seems to be the representation that allows for the quickest implementation of Date.

8.3.5 Attributes

Choosing R_1 as the basis for our implementation determines the instance fields except for their specific type. An obvious first choice is three int variables. However, days, months, and years does not behave as integers, so it is wise to define three helper classes *Day*, *Month*, and *Year* on which we, by need as we go along, will define the proper behaviour. Arguments for this kind of "extreme decomposition" are provided by Fowler in *When to Make a Type* [Fowler 2003]. The three helper classes, which come into existence due to refinement and decomposition, may be encapsulated as inner classes of class *Date*, since they are not relevant to the surroundings. Their existence is merely a result of our choice of representation for class *Date*.

The extreme decomposition into classes *Day*, *Month*, and *Year* enables independent and incremental development and testing of each of the four component classes, as well as each of the methods in the classes. For people who at a later stage must read the code, the decomposition supports independent and incremental comprehension of the individual parts. Extreme decomposition is beneficial for software developers as well as for watchmakers (recalling Simon’s parable about Hora and Tempus in section 2.1.1).

The definition of class *Date* after adding the fields is presented in Figure 8-6. The method stubs are unchanged. Comments from previous code segments are left out for brevity; only comments for new methods are included from here on.

```
class Date {
  private Day day;
  private Month month;
  private Year year;

  /**
   * Constructor: a date instance with an arbitrary
   * (fixed) value.
   */
  public Date() {
    day= new Day(10);
    month= new Month(3;)
    year= new Year(2007);
  }

  public void setToNextDay() {}
  public String toString() { return null; }

  private class Day {
    private int d;  // 1 ≤ d ≤ month.days()
    public Day(int d) { this.d= d; }
  }

  private class Month {
    private int m;  // 1 ≤ m ≤ 12
    public Month(int m) { this.m= m; }
    public int days() {}
  }

  private class Year {
    private int y;
    public Year(int y) { this.y= y; }
  }
}
```

Figure 8-6: *Adding instance fields and constructor to Date and its private inner classes*

8.3.6 Methods

The next step is to implement and test the methods. Some methods may be easy to implement in one step; *toString* in our example falls into this category. Other methods may require more work. In this case, partial solutions may be used for initial versions. Figure 8-7 shows our class after implementing function *toString* and a first, naïve version of *setToNextDay*. These refinements generate the need for a method in class Day and *toString* methods on all three helper classes. Additions and modifications compared to the former version of Figure 8-6 are marked in bold face.

```
class Date {
  // ...

  public void setToNextDay() { day.inc(); }

  public String toString() {
    return year + “-” + month + “-” + day;
  }

  private class Day {
    private int d;  // 1 ≤ d ≤ month.days()
    public Day(int d) { this.d= d; }
    public void inc() {
      d= d + 1;
    }
    public String toString() {
      if ( d < 10 ) {
        return “ ” + d;
      } else {
        return “” + d;
      }
    }
  }

  private class Month {
    private int m;  // 1 ≤ m ≤ 12
    public Month(int m) { this.m= m; }
    public int days() {}
    public String toString() {
      if ( m < 10 ) {
        return “ ” + m;
      } else {
        return “” + m;
      }
    }
  }

  private class Year {
    private int y;
    public Year(int y) { this.y= y; }
    public String toString() { return “” + y; }
  }
}
```

Figure 8-7: *Naive implementation of Date*

This partial solution is indeed a very naïve implementation. Nevertheless, as discussed in section 7.1.5, we might claim that method *setToNextDay* is 97% correct since it works correctly in 353 out of 365 cases!

The version in Figure 8-7 would benefit from refactoring by providing a single abstraction to handle string representations of integers of only one digit, but we ignore this issue here.

Incrementing field *d* in method *inc* of class *Day* might violate the representation invariant of the class, $1 \leq d \leq$ *month.days*(); in this special case, the above implementation of *inc* fails to work properly. We have to check for this special case and handle it appropriately. For simplicity, we temporarily assume 30 days in every month.

In the special case where *day* after being incremented exceeds the number of days in the month, *day* must be set to 1 and field *month* must be incremented. Following our *Special Case* rule from section 2, we deal with this special case by introducing a new private method, *checkOverflow*. Now, incrementing the variable *month* might also violate the representation invariant; this special case is handled similarly by introducing a private method *checkOverflow* in class *Month*, which is called after incrementing *month*. Except for the assumption of 30 days in every month, the program is now complete (total and consistent). Figure 8-8 shows the resulting code.

```
class Date {
  // ...

  private class Day {
    private int d;  // 1 ≤ d ≤ month.days()
    public Day(int d) { this.d= d; }
    public void inc() {
      d= d + 1;
      checkOverflow();
    }
    private checkOverflow() {
      if ( d > month.days() ) {
        d= 1;
        month.inc();
      }
    }
    // ...
  }

  private class Month {
    private int m;  // 1 ≤ m ≤ 12
    public Month(int m) { this.m= m; }
    public int days() { return 30; }
    public void inc() {
      m= m + 1;
      checkOverflow();
    }
    private void checkOverflow() {
      if ( m > 12 ) {
        m= 1;
        year.inc();
      }
    }
    // ...
  }

  class Year {
    // ...
    public void inc() { y= y + 1; }
  }
}
```

Figure 8-8: *Partial implementation of Date*

To complete our implementation according to the full requirements, we have to replace the literal 30 with the correct number of days in every month. To calculate the number of days in the current month, we declare a local array variable in this method to hold the number of days per month (with 28 days for February), and the method returns the number of days in the current month by looking up the number in the array. This brings us almost to the finish line: the implementation now works except for the case that the current year is a leap year (“99.93% correctness”, section 7.1.5), see Figure 8-9.

```
class Date {
  // ...
  private class Month {
    // ...
    public int days() {
             //  1  2  3  4  5  6  7  8  9 10 11 12
      int t= {0,31,28,31,30,31,30,31,31,30,31,30,31};
      return t[m];
    }
  }
  // ...
}
```

Figure 8-9: *Improved implementation of Month.days*

A leap year is a special case of *Month.days*; according to the special case rule we introduce a new private method to deal with it. In this case, we introduce a boolean method *Year.isLeapYear* that returns true if the current year is a leap year. The implementation of this method is a straightforward implementation of the leap year rule: a year is a leap year if the year is divisible by 4 but not by 100 or if it is divisible by 400.

The hardest part of this calculation is the check whether a number can be divided by another, so, again following the Mañana Principle, we use a method *divides* that gives us the result and then implement that method later. The additional aspects of the complete solution is presented in .

```
class Date {
  // ...
  private class Month {
    // ...
    public int days() {
             //  1  2  3  4  5  6  7  8  9 10 11 12
      int t= {0,31,28,31,30,31,30,31,31,30,31,30,31};
      return t[m];
    }
  }

  class Year {
    // ...
    public boolean isLeapYear() {
      return ( isMultipleOf(4) && !isMultipleOf(100) )
            || isMultipleOf(400);
    }
    private boolean isMultipleOf(int x) {
      return y % n  ==  0;
    }
  }
}
```

Figure 8-10: *Final solution for class Date*

This completes our detailed development of classes *Date*, *Day*, *Month*, and *Year* according to STREAM.

8.3.7 Discussion

The above development of a class implementing *Date* demonstrates the application of the techniques set out in section 8.2. The most relevant observation is that every step is broken into small, manageable chunks.

Some of the steps in our technique are fairly easy to learn (creating method stubs, defining the instance fields after deciding on a representation); others require much practice (creating tests, implementing methods).

The detailed discussion of the method implementation has demonstrated that the harder parts of a programming task can be decomposed using a technique that integrates extension and refinement as described in chapter 7.

8.4 Graspability of STREAM

A number of vital aspects contribute to make STREAM accessible and graspable to novices: (1) the cognitive apprenticeship approach revealed through worked examples provided as live programming in class or as videos; (2) careful guidance in exercises and assignments; (3) careful characterization of the process and its elements in written course material; and (4) practice by the students.

The cognitive apprenticeship approach is described in section 8.1.3, and our phrasing of exercises and assignments as guide tours through incremental development processes is exemplified in section 8.1.2. As for the characterization of the process, we have carefully chosen mnemonic names, e.g. STREAM, TEACH, and the Mañana Principle as well as the specific rules. These examples of chunking and categorizations help reduce the cognitive load such that the programming method does not “get in the way” by occupying too much valuable working memory. The programming method can be grasped incrementally (e.g. by first remembering the steps and the Mañana Principle but not quite remembering the specific rules), and it is our experience that it supports students from the moment they become aware of it. And once grasped, the method constitutes valuable *knowledge schemas* and *skill schemas*, which improves learning and transfer (section 3.1.1). Or, to put it differently: The build-in scaffolding of STREAM, which explicitly mirrors the structure of the complex task of implementing a class through extension and refinement of its methods, supports schema creation and transfer and thereby conveys a method for accomplishing similar tasks in the future [Rosson et al. 1996].

8.5 Conclusion

In section 8.1 we characterised the program development process as the challenge of findings one’s way through an n-dimensional space of extension, refinement, refactoring, test-case development, optimization, styling, etc.

Traditionally, we do almost nothing to guide novices through this space. In section 8.1.1, the current state of affairs is characterized as invitations for random walks, since the students are not provided concrete guidance about program development.

We have presented our approach to programming education, which draws upon the results of cognitive load theory and offers an alternative to endless random walks. Instead, we suggest guided tours. By providing guidance and scaffolding with respect to all dimensions involved, we can ensure that students exercise the important aspects of programming while keeping the cognitive load within the bounds where learning outcome is optimized.

Our primary means of providing guidance with respect to incremental development is through the structure of the teaching material (textbook, exercises and assignments, and videos) and an apprentice-based teaching approach.

Guidance with respect to refinement is provided through a carefully designed novice's process of object-oriented programming: STREAM. The process is a carefully down-scaled version of a full and rich software engineering process particularly suited for novices learning object-oriented programming. In using it, we achieve two things: help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the aspects of software engineering.

Besides describing the process at an abstract level, we have described the process and provided an example of its use.

Finally, we have discussed the graspability of STREAM, which is ensured through four aspects: (1) the cognitive apprenticeship approach revealed through worked examples provided as live programming in class or as videos; (2) careful guidance in exercises and assignments; (3) careful characterization of the process and its elements in written course material; and (4) practice by the students.

We conclude that we have provided elaborate and comprehensive answers to the two research questions addressed in this chapter: $Q_{5.1}$: How can we down-scale modern software development methods to a programming process for novices? and $Q_{5.2}$: How can we structure the relevant body of knowledge so that it can be most readily grasped by the learner?

9 Instructional Design

In this chapter, we address the remaining parts of research question 5:

$Q_{5.3}$: How can we organize efficient learning paths/courses that incrementally approximate best-practice in modern software development at the level of novices?

$Q_{5.4}$: How can we adopt results of cognitive science and educational psychology to the instructional design of introductory programming education?

Section 9.1 presents some fundamental principles of programming education that guides us in organizing a course. In section 9.2, we discuss the overall organization of an introductory programming course that incrementally approximates best-practice of modern software development. In doing so, we apply results of cognitive science and educational psychology in general and cognitive load theory in particular to ensure an instructional design of an introductory programming course that balances the cognitive load in order to optimize learning. In section 9.3, we report on our experience in applying the approach. Section 9.4 reports on related work. Section 9.5 is the conclusion of the chapter.

Parts of the work reported in this chapter were carried out with Christensen, Bennedsen, Alphonce, and Decker and is described in the chapters 17-21 in the second part of the dissertation. The incorporation of cognitive science and educational psychology is my own work, as is the description of course organization to incrementally approximate best-practice of modern software development; none of this has yet been published.

9.1 Principles of programming education

An instructional design takes as its starting point —or specification— the syllabus for the course to be designed. As in the case of program development, it is wise to break down the goals of the course into a sequence of simpler or weaker goals that ultimately leads to the requirements of the course.

When making decisions about course design, we all apply some kind of axioms or values, typically implicit and unspoken, which determine *how* we organize a course. Instructional design is practiced along the full continuum ranging from educators doing things in a certain way because "this is the way we always have done it, and it works —those who survive are good", to the other extreme where educators carefully design their courses according to established pedagogical models, principles, and techniques.

To guide the instructional design of the introductory programming course at DAIMI, we lean on nine principles that permeate the organization of activities at all levels of course design. The principles are:

1. Consume before produce

2. Present worked, exemplary examples
3. Reinforce specifications
4. Reveal process and pragmatics
5. Provide hands-on opportunities
6. Define progression in terms of complexity of tasks
7. Reinforce patterns and conceptual frameworks
8. Ensure constructive alignment
9. Provide care and support

Many of the principles mutually support each other and integrate well with the overall pattern- and apprentice-based approach to instruction. In the following, we discuss the nine principles and how they unfold in the context of an introductory object-oriented programming course.

9.1.1 Consume before produce

In [Pattis 1990], the author introduces the *call before write* approach to teaching introductory programming, arguing that it "allows students to write more interesting programs early in the course and it familiarizes them with the process of writing programs that call subprograms; so it is more natural for them to continue writing well structured programs after they learn how to write their own subprograms". Pattis points out that the "call before write" approach requires the linguistic ability to cleanly separate a subprogram's specification from its implementation.

In [Meyer 1993], the author introduces the notion of the *inverted curriculum* as follows: "This proposal suggests a redesign of the teaching of programming and other software topics in universities on the basis of object-oriented principles. It argues that the new 'inverted curriculum' should give a central place to libraries, and take students from the reuse consumer's role to the role of producer through a process of 'progressive opening of black boxes'". Currently, Meyer and Pedroni are implementing the inverted curriculum at ETH [Pedroni 2003, Pedroni et al. 2006].

In [Schmolitzky 2005], the author briefly mentions the notion of *consuming before producing* by providing three specific examples. One example is: "BlueJ allows beginning with an object "system" with just one class where students just interactively use instances of this class (they *consume* the notion of interacting with an object via its interface). *Producing* the possibility of interacting with an object, on the other hand, requires more knowledge about class internals and should thus be done after the principle of interaction with objects is well understood".

We rely heavily upon the principle of *Consume-before-Produce*. The principle is applicable to a wide number of topics, e.g. code, specifications, class libraries, design patterns, and frameworks.

Code: We employ the principle with respect to the way students write code at three levels of abstractions: method level, class level, and class model level as follows: (1) *Use methods* (as indicated above, BlueJ allows interactive method invocation on objects without writing any code). At this early stage, students can perform experiments with objects in order to investigate the behaviour and determine the actual specification of a method. We return to this issue in section 9.2.3. (2) *Modify methods* by altering statements or expressions in existing methods. (3) *Extend methods* by writing additional code in existing methods. (4) *Create methods* by adding new methods to an existing class. This may also be characterised as *extend class*. (5) *Create class* by adding new classes to an existing model. This may also be characterised as *extend model*. (6) *Create model* by building a new model for a system to be implemented.

Specifications: Specifications and assertions can be expressed in many ways, e.g. as Javadoc, test cases, general assertions in code, loop invariants, class invariants, and system invariants (constraints in the class model, for instance a specific multiplicity on a relation between two classes). In all cases, students are gradually exposed to reading and comprehending specifications prior to producing specifications themselves.

Class libraries: Not many years ago, the standard syllabus for introductory programming courses encompassed implementation of standard algorithms for searching and sorting as well as implementation of standard data structures such as stacks, queues, linked lists, trees, and binary search trees.

These days, standard algorithms and data structures are provided in class libraries, ready to be used by programmers. By using class libraries that provide advanced functionality, students can do much more interesting things more quickly. Also, experience as consumer presumably motivates learning more about principles and theory behind advanced data structures and packages for distributed programming, etc.

Consequently, algorithms and data structures is one of the areas where we can sacrifice material in order to find room for all the new things that make up a modern introductory programming course.

Design patterns: Design patterns can easily be (partially) covered in the introductory programming course if the Consume-before-Produce principle is applied. We employ a progressive approach to design patterns that covers six steps: (1) *Use it*. The students should gain an appreciation of the usefulness of a pattern before using an implementation of it. For example, when learning the Strategy pattern, students should gain experience by using a strategy by, say, sorting the elements of a *List* using a *Comparator*. (2) *Conceptualize it*. Students should be engaged in a discussion of the general architecture of a pattern. For example, when learning the Strategy pattern, students must come to understand the concept of a strategy and the opportunities it provides for dynamically changing object behaviour. (3) *Build it*. The next gain in understanding comes from implementing a pattern. When learning the Strategy pattern, students must create a class with an associated strategy. (4) *Analyze/study high quality code*. A deeper understanding of any pattern comes from studying a variety of high quality implementations of the pattern. In the case of the Strategy pattern, it is perhaps at this point that the students begin to truly grasp the beauty and flexibility of factoring strategies out into separate associated classes. (5) *Design and construct*. Students must at some point apply their knowledge of patterns to design and construct software components with several interacting design patterns. (6) *Evaluate*. A final step in the process of learning to use design patterns comes in being able to evaluate and critique the use (or lack of use) of design patterns in software.

In the introductory programming course, it is feasible to reach step 3 above. In the paper in chapter 20, a more thorough discussion of educational aspects related to teaching design patterns.

Frameworks: Sometimes even using a piece of software can be a daunting task. Frameworks are examples such complex pieces of software.

Frameworks constitute a part of our introductory course, but in order to ease comprehension of a complex framework like Swing, we provide a stepping stone in the form of a small and simple framework, which students consume by making a few simple instantiations. Then we provide a general taxonomy for frameworks, which is easily understood and grasped in the context of the simple toy framework. With the concrete experiences and the taxonomy in the bag, students are now mature to embark on using more complex frameworks such as Swing.

In section 9.2.9 and chapter 18, we provide a more thorough discussion of our approach to teaching frameworks in the introductory programming course.

9.1.2 Worked, exemplary examples

Worked examples play a key role in our course. As mentioned in section 8.1.2, the textbook we apply is problem-based, and this approach is followed up by exercises, assignments, and worked examples of program development performed live in class or provided as videos (see section 8.1.3).

Examples are considered very important for learning in general. Novice programmers even think they learn programming best from examples [Lahtinen et al. 2005]. However, computer science educators use many examples that might do more harm than good (see for example [Holland et al. 1997, Hu 2005, Malan et al. 2004, Westfall 2001]). It is therefore mandatory that these examples are not only correct but can also serve as a template for "good" design and style in any reasonable aspect.

Exemplary can mean many things depending on purpose, perspective, and point of view. Our concern is to address the topic from a didactical/pedagogical perspective.

All examples must follow all the definitions, and "rules" we have introduced, i.e. we must say as we do and do as we say. This requires very careful planning and development. According to our own experience, shortcuts or examples constructed "on-the-fly" will almost certainly introduce unintended problems.

Consequently, follow accepted principles, rules and guidelines. However, make sure to keep the focus on OOP novices. Many principles, rules, and guidelines are targeted toward professionals. They might not be applicable or even meaningful for novices. Accepted principles, rules, and guidelines encompass (1) general coding guidelines and style, like naming of identifiers, indentation, categorization of methods, like accessors, mutators, etc.; (2) common principles, like the ones summarized in [Martin 2003, Meyer 1997]; and (3) object-oriented design heuristics, like the ones described in [Gibbon et al. 1996, Riel 1996].

Finally, it pays off to get to know your students to be able to give them relevant and challenging examples. In courses such as ours, where students come from a large number of study programmes, it is vital to ensure that the examples are meaningful to all. With help from faculty members of other departments, we have developed subject specific assignments targeted at students from all study programmes that officially include the introductory programming course.

9.1.3 Reinforce specifications

As evident from chapter 7, specifications (in one form or another) play a key role in incremental program development, and as Pattis points out (section 9.1.1): “the linguistic ability to cleanly separate a subprogram’s specification from its implementation” is required in order to practice the “call before write” approach. We therefore hold as principle that the notion of specification is treated as a first-class citizen in the introductory programming course.

In a recent study [Lister et al. 2006], students’ program comprehension abilities were analyzed according to the SOLO taxonomy (Structure of Observed Learning Outcome) [Biggs 2003]. In a conclusion the authors write: “Teachers also need to test their students in a way that is intended to elicit a relational response. In providing such a response, a student manifests an ability to read several lines of code and integrate them into a coherent structure — to see the forest, not just the trees”. An example of a question to elicit a relational response from the students is the following:

In plain English, explain what the following segment of Java code does:

```
bool bValid= true;

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

Figure 9-1: *An “explain in plain English” question*

Only 33% of the students manifested a relational response to the question, and the authors write that this result “may illustrate why the weaker students in many CS1 classes struggle to write code”.

Code comprehension, however, is a very difficult task if no guidance is provided. Consider, as a challenge, the following segment of Java code (assuming that function *min* returns the minimum value of its two parameters):

```
int[] b= { ... }; // an arbitrary sequence of integers
int N= b.length, n= 0, s= 0, r= 0;

n= 0; s= 0; r= 0;
while ( n != b.length ) {
    r= min(r+b[n], 0);
    s= min(s, r);
    n= n + 1;
}
```

Figure 9-2: *Challenge program*

The program in Figure 9-1 is easily comprehensible for an expert. Our conjecture is that this is not because experts generally are exceptionally good at code comprehension; it is only because we have seen (the pattern in) this code so many times that we have a conceptual schema that makes it trivial to comprehend the program. Few people have a conceptual schema to aid comprehension of the program in Figure 9-2, and we assume that most readers, although being expert programmers, will have a hard time comprehending this —on the surface— simple program segment.

We suggest an alternative interpretation of the result found by Lister et al. Since code comprehension indeed is a difficult task, it should not be left to chance. We must explicitly aid code comprehension by providing specifications that can act as beacons in the effort of comprehending code segments. If we add a specification and a loop invariant, it becomes trivial to comprehend and appreciate the behaviour and structure of the challenge program —provided of course that the reader is familiar with specifications and loop invariants (see Figure 9-3).

```
int[] b= { ... }; // an arbitrary sequence of integers
int N= b.length n= 0, s= 0, r= 0;

// req: [true, s = min segment sum of b[0..N[ ]

n= 0; s= 0; r= 0;
// inv: s = minimal segment sum of b[0..n[  ∧  0 ≤ n ≤ N  ∧
//      r = minimal segment sum for a segment ending in n-1
while ( n != N ) {
    r= min(r+b[n], 0);
    s= min(s, r);
    n= n + 1;
}
```

Figure 9-3: *Challenge program with annotations to ease comprehension*

This is just one example of the key role that specifications have in programming. We have just started utilizing the potential of the conceptual framework provided in chapter 7, but we consider it to have great potential for the way we educate novices in the skills of programming. We return to this issue in chapter 10.

9.1.4 Reveal process and pragmatics

Our emphasis on the programming process and the cognitive apprentice approach permeates this whole dissertation; we shall neither repeat previous statements nor expand further on the subject but just once again point out the importance of "taking into account the implicit processes involved in carrying out complex skills when teaching novices" (see section 8.1.2 and 8.1.3).

9.1.5 Hands-on

In [Decker et al. 1993], the authors writes: "It seems incredible that it has taken us 20 years to recognize that that programming is best taught as a 'contact sport'. Students, we now know, learn best through directed hands-on interaction with the computer and by reading meaningful working programs that provide a context for language learning".

We aim at organizing most if not all study activities such that hands-on experimentation is a required part of the activity. The book we use suggests hands-on activities while reading the text; almost all exercises and assignments contain a practical element of programming; there are weekly mandatory programming assignments; and recently we changed the format of instruction such that all TA hours (four a week) are in a computing lab that, if the need arise, momentarily can be turned into an ordinary class room We used to have only one lab hour per week and three hours of theoretical instruction, but student performance as well as feedback from evaluations demonstrate that hands-on activities (organized as pair-programming) are much more effective.

9.1.6 Progression in terms of complexity of tasks

Typically, progression in introductory programming courses is dictated by a bottom-up treatment of the language constructs of the programming language being used, and this is the way most textbooks are structured. We hold as a general principle that the progression in the course is defined in terms of the complexity of the worked examples presented to the students and the corresponding exercises and assignments.

In [Schmolitzky 2005], the author sketches eleven complexity levels of object systems used in software engineering education. The levels range from single class programs to programs with several packages of classes.

As described in section 7.2, we employ a model-driven approach to programming, which provides a nice framework for characterizing the complexity of a programming task. The progression is as follows: (1) *Single class*. We start out with programs consisting of just one class. (2) *Simple recursive relations*. We then introduce relations between classes which, in the case of only one class, reduces to recursive relations, such as descendant and ancestor, of 0..1 arity. (3) *General recursive relations*. Next, we generalise to relations of 0..n arity, which nicely motivates the need for generic collections and loops. (4) *Two classes*. Next (or simultaneously), we introduce examples of two classes while exploiting the now well-known relations between classes. (5) *Interfaces*. We then introduce interfaces and subtype polymorphism, laying the ground for exploiting generic algorithms in the collection framework by implementing interfaces such as *Comparable* and *Comparator*. (6) *Inheritance*. Next we introduce the inheritance relation and the notion of abstract classes. We now have all model elements at our disposal, and we can use these to provide arbitrary complex class models.

As suggested by the theory of cognitive science and educational psychology, we provide several examples of each generic class model (say two classes related by an association of 0..n arity), we vary the specific form of the problem type, we gradually increase the functional complexity of the examples, we alternate examples and practice problems, and we gradually decrease the amount of guidance provided. All these initiatives have the primary purpose of optimizing learning. See section 3.3 (the variability effect) and section 3.4 (worked examples and cognitive skill acquisition).

9.1.7 Reinforce patterns and conceptual frameworks

The fundamental motivation for a pattern-based approach to teaching programming is that patterns capture chunks of programming knowledge. According to cognitive science and educational psychology, explicit teaching of patterns reinforces schema acquisition as long as the total cognitive load is “controlled”; see section 3.1, 3.3, and 4.2.3 (*Patterns for schema creation*).

We reinforce patterns at different levels of abstraction including elementary patterns, algorithm patterns, and design patterns, but equally important, we provide a conceptual framework for object-orientation that qualifies modeling and programming and increases transfer [Knudsen et al. 1988, Madsen et al. 1993, ch. 18]. Furthermore, we stress coding patterns for standard relations between classes.

9.1.8 Constructive alignment

Constructive alignment was developed by Biggs and is well documented in [Biggs 2003]. In constructive alignment, according to Biggs, we first state the learning outcomes we intend our students to achieve. The outcome statements contain a learning activity, a *verb*, that students need to perform to properly achieve the outcome. That verb (e.g. ‘explain’, ‘apply’, ‘reflect’) then needs to be activated by the teaching and learning activities we give students: lecturing to them usually doesn't do that. The assessment tasks should also require students to enact that same verb. How well they solve those problems is the authentic assessment, not sitting exams about what we have taught. That verb is what achieves alignment: it is in our intended outcomes, in the teaching and learning activities, and in the assessment tasks. Traditionally, educational systems are not aligned. The curriculum is usually a list of topics telling teachers what to ‘cover’, the default teaching method is the lecture, in which students are told about the topic —they do not have to *enact* their understanding. Memorising material to report back in an exam likewise rarely requires students to put their understanding to work. The SOLO taxonomy helps to map levels of understanding that can be built into the intended learning outcomes. The DVD *Teaching Teaching & Understanding Understanding* provides an excellent introduction to the theory of constructive alignment [Brabrand 2006].

Four years ago, we altered the syllabus of the introductory programming course rephrasing the intended learning outcome in terms of the SOLO taxonomy; we altered the examination form, the contents, and the instructional design aiming at an aligned course. It has been a positive experience. Our efforts are documented in detail in chapter 21. We briefly return to the topic in section 9.2.8.

9.1.9 Care and support

The last principle to be mentioned is caring. Showing attention to the students and demonstrating that you, as educator, care about their progression and learning outcome of your teaching have a tremendous effect on motivation.

Support can be provided in many ways: by being available at classes and taking the time to talk to students, answer their questions, and pose questions to them and by providing opportunities for help and guidance, e.g. 1-1 sessions, group sessions, TA support at lab sessions, and alert responses to postings on discussions boards.

The main issue is to convey seriousness and true interest in the learning outcome of the students.

9.2 A model-driven approach to OOP

In this section we discuss the overall organization of our introductory programming course. The course is characterised as model-driven because the course progression is defined in terms of increasingly complex specification models, not constructs of the programming language as is the custom. Of course we teach about the programming language and relevant parts of the class library, but these things are treated by need rather than being the primary driver of the progression of the course (for details, see Table 9-1).

In designing the course, we have applied results of cognitive science and educational psychology in general and cognitive load theory in particular to ensure an instructional design that balances the cognitive load in order to optimize learning.

The course lasts seven weeks (one quarter) with four lecture hours and four lab hours with a TA per week. There are weekly mandatory assignments except for the first week of the course. The final exam is in week eight or nine. The course supposedly takes up one third of the students' time of the quarter.

Figure 9-4 provides an overview of the course; in the following sections we expand on each of the six phases of the course. The first section is a presentation of the course goal. The following six sections correspond to the six phases described in Figure 9-4. Section 9.2.8 is about the final exam. An extended version of the course lasts a full semester; aspects of this course are briefly discussed in section 9.2.9. Section 9.2.10 is a conclusion.

Week	Phase	Goal/Content	Instructional design
1.5	Getting started (method-use)	Overview of fundamental concepts. Learning the basics of the IDE.	Role-play, worked examples, fill in the blanks
1.5	Learning the basics (method extend and method create)	Class (access modifiers), object State (type, variable, value, integer) Behaviour (instantiation, constructor, method declaration (signature (formal parameter, return type)), method body (assignment, invoking a method, actual parameter, returning a value))) Control structures (sequence, iteration)	Worked examples, fill in the blanks (larger and larger chunks of blanks)
1	Conceptual framework and coding recipes (class-extend)	Control structure (selection, more iteration) Data structure (Collections) Class relationship (aggregation, association) Schemas for implementing structure (class relations)	Worked examples, fading guidance
1	Programming method	The mañana principle Schemas for implementing functionality (how	Worked examples, fad-

	(STREAM) (class-extend and class-create)	and in which order)	ing guidance
1.5	Subject specific assignments (class create)	Train on harder and more challenging tasks (problems) Motivation: tasks/problems are picked from the domain of the students' major subject (bio-informatics, business, chemistry, computer science, economy, geology, math, multimedia, nano science, etc.)	Guided problem solving
0.5+	Practice (class-create)	Achieve routine in solving standard tasks (UML2Java)	Drill, drill, drill

Table 9-1: *Overview of instructional design of the introductory programming course*

9.2.1 Goal

Discussing an implementation makes little sense without knowing the specification. Thus, before addressing *how* we have organized the introductory programming course, we present the course goals as described in the official syllabus:

Purpose: The purpose of the course is for students to learn the foundation for systematic construction of simple programs and, through this, to obtain knowledge about the role of conceptual modeling in object-oriented programming. The goal is that students become familiar with a modern programming language, fundamental programming language concepts, and selected class libraries.

Competencies: After the course, students should be able to use fundamental elements in a modern programming language, read specification models, implement simple specification models in a programming language, and use selected class libraries.

9.2.2 Getting started

On of the great challenges of the introductory course is how to get started. This is particularly challenging for object-oriented programming due to the many interrelated basic concepts, which cannot easily be taught and learned in isolation. Writing even the smallest meaningful class requires approximately 15 basic concepts: object, class, instantiation, constructor, method declaration, formal parameter, return type, variable, type of variable, value, integer, assignment, method invocation, actual parameter, returning a value.[36] Furthermore, the basic object-oriented concepts (class and object) represent a higher level of abstraction than in traditional procedural programming (int, char, etc.). Together, this results in a higher threshold for the learner in order to grasp the basic concepts.

Pattis has characterized the challenge of getting started as the *big bang problem* where a large number of diverse topics have to be covered —often su-

[36] Personal communication with Moti Ben-Ari.

perficially— in order to build up enough infrastructure to present full programs that do something.[37]

We address the challenge from two flanks: *conceptual overview and intuition* and *practical programming experience*. We aim at providing *conceptual overview and intuition* and intuitive understanding of the key concepts by describing an everyday scenario, which we exercise as a role-play and implement in Java. We enable *practical programming experience* from the very first day by providing a simple abstraction, which is easily graspable and where both simple and advanced programs can be expressed easily. The abstraction is a class modeling a *Turtle*. It takes as its starting point the familiar turtle graphics developed by Seymour Papert and others at MIT in 1967 [10,1]. We use it to give an intuitive introduction to concepts such as object, class, instantiation, method invocation, behaviour, object identification, state, behaviour, control flow, and actual parameter. It is surprising how many key concepts can be revealed from a simple and intuitive example such as a turtle class.

The students start as consumers, but the limited capabilities of class *Turtle* for new methods to produce more impressing behaviour than provided by the interface of class *Turtle*. So, the students almost suddenly become producers, and this enables emphasis of more fundamental concepts such as constructor, method declaration, formal parameter, variable, type of variable, value, integer, assignment, and parameterization.

Of course, the students do not fully grasp the concepts, but they understand them sufficiently well that we can commence a more detailed explanation of the basic concepts. Class *Turtle* is an excellent weapon to fight the big bang problem.

We provide plenty of worked examples, and the students either fill in the blanks or undertake more challenging tasks depending on their degree of programming experience. Some settle for simple drawings like basic geometric shapes, some draw multiple story houses and other complex objects (nested loops and procedural abstraction), and some draw colourful logarithmic spirals and fractals. In this way, class *Turtle* is a “one size fits all” solution.

Somewhat to our surprise, it turned out that package *Turtle* could play many more roles within the introductory programming course than initially anticipated: It has become a recurring vehicle for introducing such diverse topics as object models, recursion, polymorphism, class hierarchies, and frameworks. Indeed, turtles popped up here, there, and everywhere when we first started exploiting their potential.

A detailed presentation of our use of the *Turtle* class in the introductory programming course is provided in chapter 17.

---

[37] Personal communication with Richard Pattis.

9.2.3 Learning the basics

After the initial exercises where basic concepts are explored and somehow grasped (corresponding to a constructive early phase in the terminology of cognitive skill acquisition, section 3.4), we proceed with activities that aim at a more complete understanding of the fundamental concepts.

In the beginning of this phase, when the students are not yet fluent in Java programming, they experiment with objects provided by us. In order to install awareness about the notion of requirements and partial specifications, we provide objects where the actual behaviour corresponds to a specification that is weaker than the anticipated requirements. For example, we provide a partial implementation of class *Date*. The students invoke methods on *Date* objects and realize that “sometimes it works as expected, sometimes it does not”. We ask them to alternate between making experiments and refining their hypothesis about the actual specification of the method they invoke. Exercises like this one generate excitement; the students become curious and eagerly want to open the class to see of the code that behaves in this “strange” way.

The game we are playing can be characterised as finding a specification in the lattice spanned by *abort* and the expected requirements of the methods (Figure 9-4). No requirements need to be explicitly expressed, since we rely on everyday objects (e.g. *Date*) with well-known behaviour. The students generate their expectation of the requirements instantly. Through exercises like this one, the students can experiment with programs and build images of the programs behaviour and structure without writing a single line of code.

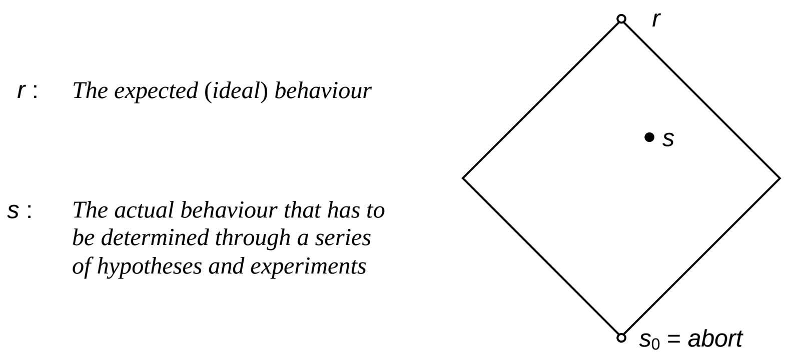

r : The expected (ideal) behaviour
s : The actual behaviour that has to be determined through a series of hypotheses and experiments
r
s
s0 = abort

Figure 9-4: *Use of hypotheses and experiments to determine behaviour*

Of course, writing code is important, but it is a separate concern. In this phase, the programming tasks are described by class models of initially one class only (first with no relations and then with associations of 0..1 and 0..* arity) and then two classes (again with various relations between them). Specific examples are *Clock*, *Die*, *Heater*, *Person* (with ancestors), *ClockDisplay* and *NumberDisplay* (a *ClockDisplay* with two *NumberDisplay* objects), *DieCup* and *Die* (a *DiceCup* with two *Die* objects), etc. The generic models the students learn to implement in this phase are sketched in Figure 9-5.

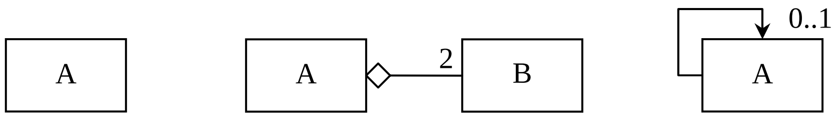


Figure 9-5: *Generic class models in the learning-the-basics phase*

For each kind of generic class model, we reinforce its standard implementation in Java.

We introduce the generic models inductively, i.e. we introduce a couple of worked examples which the students finalize before generalising to the generic models.

9.2.4 Conceptual framework and coding recipes

In this phase of the course (week three) we introduce a subset of the conceptual framework for object-orientation developed by Knudsen et al. [Knudsen et al. 1988, Madsen et al. 1993]. According to Madsen et al., the object-oriented perspective on programming is defined as follows: "A program execution is regarded as a physical model simulating the behaviour of either a real or an imaginary part of the world". From the object-oriented perspective, concepts are modeled as classes and phenomena as objects. A basic understanding of phenomena, concepts, and abstraction forms the basis of the conceptual framework that provides well-defined characterizations of classification, aggregation (decomposition), and generalisation (specialisation) as ways of forming concepts from phenomena or other concepts. Object-oriented programming languages support these abstractions mechanisms in different but similar ways; thus, the conceptual framework provides knowledge and understanding that carries across different object-oriented programming languages.

The conceptual framework provides guidance for a disciplined use of components in modeling languages (e.g. UML) and abstraction mechanisms in object-oriented languages. We supplement this guidance with coding recipes for the fundamental types of relations between concepts (classes): generalisation/specialisation, aggregation/decomposition, and association.[38] In popular terms, generalisation is known as *is-a*, aggregation as *has-a*, and association as *x-a* for any verb *x* different from *is* and *has*.

In this phase, the programming tasks are described by class models built by components from a *DiceCup* with an arbitrary number of *Die* objects, a *Notebook* with many *Note* objects (each with many *Keyword* objects associated), a *Playlist* with associated *Track* objects (each with associated *Picture* objects), *Account* with (various types of) *Transaction* objects, etc. The generic models the students learn to implement in this phase are sketched in Figure 9-6.

[38] The conceptual framework, the way we use it to discipline our use of language constructs, and coding recipes for the implementation of structures bare strong resemblance to the early days of structured programming with disciplined use of language constructs to provide single entry and exit points for selections and loops, and with recipes for translation of abstract control structures into the implementation language.

Again, we reinforce the standard Java implementation of the generic class models. Coding recipes for the various kinds of relations provide general solution patterns for implementing *inter-class structure* as described in section 7.2.2 (see Figure 7-13).

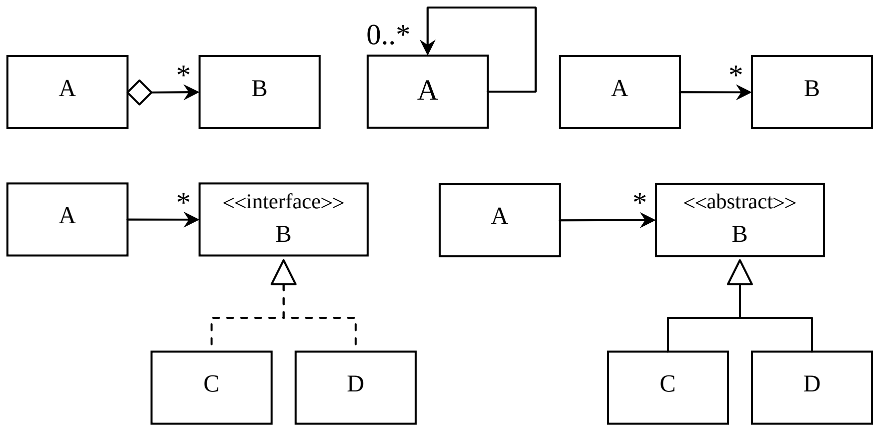


Figure 9-6: *Generic class models in the conceptual frameworks and coding recipes phase and beyond*

There are many learning-theoretic arguments for our model-driven approach to programming. We provide three:

Because of their generic nature, the abstract models directly support schema creation and transfer: “Well-designed learning environments for novices provide *metacognitive managerial guidance* to focus the students’ attention and *schema substitutes* by optimizing the limited capacity of working memory in ways that free working memory for learning. Good instruction will segment and sequence the content in ways that reduce the amount of new information novices must process at one time and, as much as possible, reinforce domain patterns to support schema acquisition and improve learning.” (Section 3.1).

Variation of form (e.g. cover story) can help novices realize that there is a many-to-one relationship between form and problem type: when students see a variety of cover stories used for identical or similar structures (of class models), they are more likely to notice that surface features are insufficient to distinguish among problem types and that problem categorization according to structural similarities (patterns) is imperative to enable reuse of solution schemas [Quilici et al. 1996]. See also section 3.4.

The modular nature of class models and the fact that the relations between classes may exist at any level of abstraction models means that they are particularly well suited for reducing cognitive load and thus improving learning: “In five studies, we provided evidence that indeed these modular examples are superior to molar examples with regard to problem-solving performance for isomorphic and novel problems, different measures of learning time, and cognitive load. The positive effects of modular examples were found regardless of the number of problem categories taught, the learning task announced, and the amount of instructional explanations given. Furthermore, modular examples proved to be superior for learners with low as well as with high prior knowledge. Therefore, the advantages of modular examples for teaching problem-solving skills seem to be very stable over a variety of instructional conditions" [Gerjets et al. 2004].

We realize that few experts actually build specification models, but still they provide an excellent overview and generic approach to (introductory) object-oriented programming. If pedagogical development tools (e.g. BlueJ and DrJava) supported integration of specification models and code, we conjecture that the effect would be even better. However, we still have that coming.

9.2.5 Programming method

After the conceptual framework and coding recipe phase, we introduce our systematic method of program development, STREAM. We have already described many teaching aspects related to this (section 8.1.2 (guidance), 8.1.3 (cognitive apprenticeship using videos), and 8.4 (graspability of STREAM), so we have only little to add here.

It is very difficult for novices to think of representation alternatives; they have trouble enough implementing what we prescribe. However, by providing representation alternatives, we can ignite discussions of implementation effort and have the students make a REM (representation evaluation matrix). Such discussions are excellent candidates for group work, and they are valuable because they make the students think and talk about representation alternatives and their consequences for method implementations *before* they start writing any code.

This important aspect of STREAM supports what we described as implementation of *intra-class structure* section 7.2.2 (see Figure 7-13). Further aspects of STREAM are described in the paper in chapter 16.

9.2.6 Subject specific assignments

More than 60% of our students are non-CS majors; they come from a broad variety of study programmes, e.g. bio-technology, business, chemistry and technology, computer science, economy, geology, mathematics, multimedia, and nano science. In the penultimate week of the course, the students work on a more complex programming task solving a problem relevant to their major. As mentioned in section 9.1.2, we have developed subject specific assignments targeted at students from all study programmes that officially include the introductory programming course. Examples of such assignments are: comparison of DNA strings, stock analysis, simulation of water flow, calendar, spell checker, image processing, and a calculator for group theory on integers.

We provide partial solutions and guiding material, and the students implement the missing parts of the programs. The better students invent extensions while the weaker students just do the basic stuff. The primary purpose is that the students experience a non-trivial program that does something meaningful and useful for them.

9.2.7 Practice

The last week of the course is reserved for repetition, and the students start practising for the final exam by solving lots of small program assignments (typically involving three classes).

For the final exam, the students must solve a simple programming assignment within 30 minutes. The time pressure makes the students practice and practice hard. They practice throughout the course because they know what is expected of them, but in the final phase the practice even harder to make sure that they have sufficiently experience to solve the assignment within the given time frame.

According to cognitive theories related to the power law of practice [Newell 1990, Newell et al. 1981], chunking of knowledge in long-term memory is the reason for improved performance when solving programming tasks. [Anderson 1993] claimed that the speedup is due to two mechanisms: Knowledge in long-term memory is converted from a slow format (declarative format or knowledge schemas) into a fast format (procedural format or skill schemas), and the speed of individual pieces of procedural knowledge also increases with practice (see section 3.4.1 on *The power law of practice* and section 3.4.2 on *Transfer*).

9.2.8 Final examination

In a recent posting to the SIGCSE mailing list, Lee wrote: “The students prefer to spend their time on passive tasks such as reading the textbook/notes and watching/listening to recorded lectures, as well as attempting theory exercises such as multiple-choice questions from past exam papers (Perhaps thinking that this will somehow help them ‘scrape through’ and make it over the pass mark in the course!) Many of these students report being able to follow and comprehend sample code given to them perfectly well, but when it comes to writing code of their own... Well, that is a whole another story!” [Lee 2007].

Lee continues: “One (somewhat cynical) way of dealing with this problem is to develop the assessment scheme in such a way that students simply cannot pass the course without writing copious amounts of original code from scratch”.

The idea of adjusting the assessment scheme is no at all cynical, on the contrary! At least it is not cynical if the goal of the course is that students *should* be able to write small (fragments of) programs on their own. According to the theory of constructive alignment, that is precisely what we should do.

As mentioned in section 9.1.8, we have altered the introductory programming course according to the theory of constructive alignment. An important goal of our introductory programming course is that the students learn a systematic approach to the development of computer programs. Having the programming process as a learning objective naturally raises the question how to include this in the assessment. Traditional assessment forms (e.g. oral or written examinations, multiple choice questions) are unsuitable to test the programming process.

We have developed a practical lab examination that to a certain extent assesses the students' programming process as well as the developed programs. The lab examination has as characteristics that it (1) provides a valid and accurate evaluation of the student's programming capabilities, (2) evaluates the process as well as the product, (3) encourages the students to practice programming throughout the course, and (4) can be used to assess approximately 150 students per day. A detailed description of the practical lab examination is documented in chapter 21.

9.2.9 Patterns and frameworks

This section describes the second part of the *extended version* of the introductory course (the semester version). The extra seven weeks allow us to address more advanced topics of object-oriented programming (inheritance, design patterns, and frameworks) as well as general topics such as recursion algorithm patterns, e.g. sweep, search, merge, divide-and-conquer, and backtracking. An example of our treatment of design patterns was presented in section 9.1.1.

Frameworks are class libraries, but with the additional characteristics of *inversion of control* (also known as the Hollywood principle: don't call us, we'll call you) and *hot spots* (also known as hooks or variability points).

Good object-oriented frameworks are unique examples of the strength of the object-oriented paradigm. Looking behind the scenes of good frameworks shows how careful modeling of domain concepts, use of polymorphism, and the use of design patterns makes a piece of software highly flexible and demonstrates the power of low coupling and high cohesion. It is simply a brilliant case study to learn from and, as such, the ultimate killer example of the use of design patterns.

But before opening the box and studying the beauty of the design of a framework, it is wise first to take the role of consumer. Frameworks are typically large and complex to use; it is so because use of a framework requires understanding of the part of the architecture that is exposed to application programmers.

The Java GUI framework (AWT or Swing) is an example of such a complicated framework. Despite the complexity of the framework, many introductory programming courses have adopted GUI programming as part of the syllabus. It is a dangerous business; the complexity of the framework, which stems from the huge set of classes and interfaces, pervasive subtype polymorphism, anonymous subclasses, event handling, etc., means that the cognitive load is high; thus, little working memory is left for learning.

Frameworks constitute a part of our introductory course, but in order to ease comprehension of a complex framework like Swing, we provide a stepping stone in the form of a small and simple framework that the students consume by making a few simple instantiations. Then we provide a general taxonomy for frameworks that is easily understood and grasped in the context of the simple toy framework. With the concrete experiences and the taxonomy in the bag, students are now mature to embark on more complex frameworks such as Swing.

In chapter 18, we provide a more thorough discussion of our approach to teaching frameworks in the introductory programming course.

9.2.10 Conclusion

We have described in detail the layout of the introductory programming course (the first quarter), and we have sketched the contents of a special extended semester version of the course. Constructive alignment, starting with a clear specification for the course, is the overall critical aspect of the design.

We characterise the course as model-driven because the progression is defined in terms of increasingly complex specification models. Programming skills for systematic, incremental program development from specifications expressed as annotated class models is the primary focus of the course.

We have argued how the course is designed according to results of cognitive science and educational psychology in general and cognitive load theory and skill acquisition in particular; the principal techniques applied are: worked examples, scaffolding, faded guidance, cognitive apprenticeship, and emphasis of patterns to aid schema creation and improve learning.

Constructive alignment provides the argument for the examination form. The tight time frame for the final examination puts pressure on the students; this makes them practice which —according to the power law of practice— enhances learning.

9.3 Evaluation of process competence

In section 8.1.2, we demonstrated how careful and detailed phrasing of an assignment can guide the students through an incremental programming process characterised by a (more or less) monotone development trace.

We have used similar phrasing for the final exam in order to ease evaluation of the students (the more control we have of their process, the easier it is to evaluate progress). This raises the question of how well we really can evaluate the students programming process: If we provide detailed guidance, how then can the students demonstrate their personal competence on this issue?

In order to evaluate the learning effect specifically with respect to process competence, we set up an experiment just prior to the previous final examination. We designed a programming task where no guidance at all was provided; the assignment consisted of a class model and functional specifications of the methods of the classes. The assignment is the second example of section 8.1.2 (without the final hint regarding *Comparable* and *Comparator*).

All students were invited to take part in a practice exam, and 38 students accepted the invitation (the students who accepted the invitation were representative of the whole population with respect to major).

Our goal was to evaluate the students programming process now that no guidance was provided. A group of TAs examined the students and took notes of their behaviour; the student/TA ratio was 3/1. The TA's were instructed to make notes of the students' programming process. In particular, they should make a note whenever a student violated the ‘standard process’ that had been taught in the course (demonstrated through live programming and several videos of worked examples).

The conclusion of the experiment was that all students followed the process they had been taught even though no guidance was provided. They developed one part of the program at a time nicely separating the different concerns of the task. There was some variation as to how frequent the students swapped between writing test code and writing production code and as to whether they wrote the test code before or after the production code. STREAM suggests writing test code before the production code, but almost all the students wrote the production code first.

Interestingly, hardly any of the students took the easy way out by implementing *Comparable* in order to get away with trivial implementations of three of the requested methods.

Immediately after the practice exam, we conducted informal interviews with groups of students. When asked about their testing behaviour (less frequent and after the fact), they responded that they did not feel the need for the test in order to implement the requested methods; they wrote the test because they had to, and not because the needed it to understand the task. It is hard to blame them on that because their behaviour mirrors expert behaviour (see section 6.3.2).

We refrain from drawing too strong conclusions from this experiment, but the result suggests that students do learn the process we teach —at least when they are exposed to familiar tasks. But, again, this is just as it is with experts as described in section 6.3.2. Winslow puts it this way: “Experts, when given a task in a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general (opportunistic) problem solving” [Winslow 1996, p. 18].

For the past four years (since we started with the current syllabus and exam), the pass rate has been 87%, 87%, 88%, and 93% (300+ students per year). This result is achieved without (serious) pain: Extensive course evaluations show that students spend less time on the course than is expected of them. In the first few weeks, they do not believe they are ever going to make it, but by steady practice, they come after it, and they do make it. And that is the way it should be

9.4 Related work

One of our foci, pattern-based instruction, is the essence of the work by Muller who also makes use of cognitive load theory for instructional design. Muller’s primary focus is on algorithmic problem solving (though in the context of object-oriented programming), and the pattern focus is on algorithmic patterns only [Muller 2005b, Muller et al. 2004].

In [Muller et al. 2005a], the authors indicate a perspective on incremental development that resembles ours: “The design of the course is especially aimed at enhancing algorithmic problem-solving activities in the context of OOD/P instruction. One such activity is based on the metaphor of Evolving

Types —classes that evolve over time and obtain new characteristics and new behaviour". And further: "Through the evolving-types guideline, students are involved in a gradually increased level of problem solving tasks while learning new programming constructs and data types". However, it is unclear whether the evolving-types metaphor indicates evolving classes (provided by the instructor) to support problem solving or problem solving aimed at producing evolving types.

9.5 Conclusions

In this chapter, we have addressed the remaining parts of research question 5:

$Q_{5.3}$: How can we organize efficient learning paths/courses that incrementally approximate best-practice in modern software development at the level of novices?

$Q_{5.4}$: How can we adopt results of cognitive science and educational psychology to the instructional design of introductory programming education?

In section 9.1, we presented a set of nine principles of programming education that permeate the organization of activities at all levels of course design. Many of the principles mutually support each other and integrate well with the overall pattern- and apprentice-based approach to instruction.

Section 9.2 is a description of a model-driven introductory object-oriented programming course designed according to the principles of section 9.1 and in the spirit of the Scandinavian school of object-orientation. According to the Scandinavian school, conceptual modeling is the defining characteristic of object-orientation, and it provides a unifying perspective and a pedagogical approach focusing upon the modeling aspects of object-orientation. The section presents an introductory object-oriented programming course based upon modeling; the progression in the course is defined by increasing complexity of class models rather than being dictated by a bottom-up ordering of language constructs which is the prevailing approach. We have argued how the course is designed to absorb results of cognitive load theory, cognitive skill acquisition, and constructive alignment.

We have discussed an experiment regarding students programming process competencies; while we refrain from drawing hard conclusions from the experiment we can state that the experiment does not contradict our positive impression of the students' absorption of the programming process we enforce.

As in the previous chapter, we conclude that we have provided elaborate and comprehensive answers to the two research questions which we have addressed in this chapter: $Q_{5.3}$: How can we organize efficient learning paths/courses that incrementally approximate best-practice in modern software development at the level of novices? and $Q_{5.4}$: How can we adopt results of cognitive science and educational psychology to the instructional design of introductory programming education?

10 Future Work

An important aspect —*perhaps the most important aspect*— of scientific work is the new questions it makes possible to conceive and express. We describe potential future work derived from this PhD study. The so-far-envisaged future work falls into four categories: books, evaluation, tools, and programming methodology. For each of the categories, we provide a short discussion of potential future research and development activities.

10.1 Books

It is an obvious task to produce a textbook (or some other media) on programming with emphasis on teaching the skills of programming as described in this dissertation. A less obvious effort, but perhaps more beneficial, is to write a book on programming education for an audience of educators, curriculum designers, and authors of programming textbooks. Both opportunities may be pursued in the near future.

10.2 Evaluation of instructional design

We have presented a framework for incremental program development; according to this, we have developed a programming method for novices and an instructional design for an introductory programming course. The instructional design is not a logical consequence of the framework; of course, other instructional designs adhering to incremental program development are conceivable.

Our experience with the instructional design is excellent (as mentioned in section 2.2.1, the failure rate is approximately 10% and students are generally very capable at programming considering the size of the course. However, we have not conducted any formal evaluation of the instructional design. It would be relevant to do so, e.g. by running controlled experiments and by applying the design at other institutions thus providing the opportunity for multi-institutional and multinational studies of the effect of (elements of) the instructional design. So far, we have indications from colleagues at universities in Israel, U.K. and U.S.A. expressing interest in testing (elements of) our instructional design.

10.3 Tools

In this section, we briefly indicate potential research and development activities regarding tool support for educating novices in the skills of programming. We discuss a notional machine workbench, tool support for STREAM, tool support for incremental program development, and the design of an educational programming language.

10.3.1 A notional machine workbench

According to [du Boulay 1989a], the difficulties of novices learning programming can be separated into five partially overlapping areas: *orientation* (finding out what programming is for), the notional machine (understanding the general properties of the machine that one is learning to control), *notation* (problems associated with the various formal languages that have to be learned, both mastering syntax and semantics), *structure* (the difficulties of acquiring standard patterns or schemas that can be used to achieve small-scale goals such as computing the sum using a loop or implementing a 0..* association between two classes), and *pragmatics* (the skill of how to specify, develop, test, and debug programs using whatever tools are available).

The notional machine is a (more or less abstract) description of the semantics of the programming language; a runtime model of programs. In [du Boulay et al. 1989b, p. 431], the authors point out that "one of the difficulties of teaching a novice how to program is to describe, at the right level of detail, the machine he is learning to control". They advocate teaching the novice about the "notional machine", that is, the conceptual computer whose properties are implied by the constructs in the programming language employed.

In [Gries et al. 2002], the authors present a memory model for use in teaching Java. It includes a notion of class and a way of drawing objects to which students can relate. It includes the frames on the call stack and the steps in executing method calls, including recursive calls. The model starts out simple and is extended as new concepts are introduced, ending up with nested and inner classes.

To some extent, BlueJ's object bench and interaction pane as well as the debugger of BlueJ and the possibility of inspecting objects support experimentation with the notional machine. However, it is only in a limited fashion possible to "play" with the notional machine.

To support learning about the notional machine, we envisage a tool like a workbench where one can put a Java program and play with it. Execution of a Java program is linear in the sense that it generates a trace, which is a sequence of (transitions and) states. We would like to be able to control the execution of a program while the object model is appropriately visualized (say, as an object diagram), but in a much more advanced fashion than in an ordinary debugger where one can stop execution at a break point and then stepping forward from there. We want to be able to go back in time!

We want to be able to

- stop the program in a state, say S_n
- wind/rewind to any state S_i, $0 \le i \le n$
- modify the current state (any state in the sequence)
- continue execution from a (new) state

The possibility of modifying a state can be described as a displacement from state S_k to state S_k' (see Figure 10-1).

$$\begin{tikzcd}
S_0 & S_1 & \dots & S_k \arrow[d] & S_{k+1} & \dots & S_n \\
 & & & S_k' & & &
\end{tikzcd}$$

Figure 10-1: *Modification of a state*

We think of displacement is an irreversible action, i.e. it should not be possible to rewind back in time “through” a displacement. Thus, we consider a new state generated by a displacement to be the initial state of a new execution trace yet to be generated. In other words: S_k’ is a point of no return (imagine shaded wind/rewind buttons). In the new state S_k’, the only possible action is to resume program execution or to perform yet another displacement.

Conceptually, program execution is different from winding (although, of course, winding could be implemented as execution).

Displacement should be unconstrained, i.e. it should be possible to alter the state of any variable (primitive variables as well as object references).

A workbench like this would be a great and easy-to-use tool for students for finding answers to all sorts of ‘what if’ questions. We do not think of the tool as a debugger —a tool for finding errors— although it would be superior to any debugger we know in the sense that it is possible to go back in time. That would indeed be a very efficient way of finding program errors.

For forward and backward navigation in the current execution trace, we envisage a search tool where it is possible to specify the state(s) one is searching for *explicitly*, i.e. via a predicate describing the state (e.g. “$x \geq y$” or “*s.contains*(*x*)”) or *implicitly*, i.e. via a description of the transition leaving or entering the state (e.g. “assignment to *x*” or “call of *getSalary*()”). Clearly, a search criterion may be satisfied by more than one state in the execution trace; in that case it should be possible to step through these states with suitable navigation controls (e.g. previous and next buttons) as in an ordinary text searching tool. A search tool like this might have a positive effect on the students’ ability to think in terms of states as a supplement to the prevailing habit of operational reasoning.

10.3.2 Tool support for STREAM

We know that tool or language support of a concept enhances its learning. It is therefore relevant to investigate the possibility of supporting STREAM in an educational IDE such as BlueJ.

Automatic generation of a class with stub methods from a specification (an interface or a UML class description) is a trivial example.

Support for specification models expressed as class diagrams with automatic synchronization of model and code is another example. The challenge, of course, is to keep the tool simple while providing support for things that enhance learning.

Support for the mañana principle by semi-automatic generation of a stub method (or class) when decomposition is applied is yet another example, and so is support for creating a representation evaluation matrix (REM) for an interface.

10.3.3 Tool support for incremental program development

In the previous section we briefly discussed support for STREAM in its current form. As pointed out several times, STREAM provides guidance with respect to refinement but not yet with respect to extension — improvement of the specification. However, it is relevant to investigate the possibility of providing tool support for an incremental development process.

As in the case of program *execution*, it would be tremendously useful for developers as well as educators and students to have an educational IDE that supports winding and rewinding through the *development* trace of a program.

In [Dahl et al. 1972], Dijkstra writes: "If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other [...] my point is that if we have our grip on the program text primarily as on a linear sequence of symbols, the task to establish and to describe what has to be modified tends to become prohibitively difficult as the text get longer and longer." Modern development environments provide support for various kinds of transformations on program texts such as refactoring, but, fundamentally, programs are still conceived of and manipulated by people as text. A simple tool that smoothly allows traversal through different versions of a program would be a major leap forward from current practice.

As described in section 8.1.3, we provide videos to illustrate details of the programming process. Videos are very useful to provide detailed information of the programming process at a 1:1 scale. However, it would be useful to have an IDE in which it is possible to wind/rewind through well-defined states of the development trace (e.g. states where the program is consistent).

In order to convey information about the development process of a program, it would be useful to be able to provide programs for the students where they can browse through the development trace of the master's solution. Likewise, it would be extremely useful for a teacher to be able to browse through the development trace of a student's solution in order to test and provide feedback to the student regarding their programming process. It is custom only to evaluate and provide feedback with respect to the product; however, evaluation and feedback with respect to the process is equally important if we truly want to make development of programming skills an equal goal of programming education.

10.3.4 An educational programming language

Current mainstream object-oriented programming languages are far from ideal as educational languages. C++ was a disaster; compared to C++, Java is a beauty. Despite the fact that an increasing number of institutions are moving to adopt Java in their introductory curriculum, those institutions do not by any means report universal satisfaction with Java as a teaching language. The problems that arise in using Java at the introductory level were analyzed in more detail by Roberts [Roberts 2004a], which concluded that the observed difficulties in teaching Java are representative of a more general challenge facing computing science education. As its essence, the challenge arises from two self-reinforcing characteristics of modern programming languages that have a profoundly negative effect on pedagogy:

- *Complexity*. The number of details that students must master, particularly in the application programmer interfaces (APIs) supplied along with the language itself, has grown much faster than the corresponding number of high-level concepts.

- *Instability*. The languages, APIs, and tools on which introductory computing science education depends are changing more rapidly than they have in the past.

To address the problems involved in using Java at the introductory level, the ACM Education Board initiated the ACM Java Task Force in the fall of 2003 with the following general charter: “To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity” [Roberts 2004b]. The result of the ACM Java Task Force is now available [Roberts et al. 2006].

The efforts of the ACM Java Task Force have improved the usability of Java considerably. However, Java is still a complex language with an obscure syntax and a prohibitively complicated language specification of 649 pages that adds unnecessary complexity to the task learning programming [Gosling et al. 2005].

In the terms of cognitive load theory, mainstream programming languages, including Java, contribute a tremendous amount of extraneous cognitive load to the learning process. It would be interesting to design a new educational programming language in which the primary design criterion is to minimize extraneous cognitive load when (1) learning the language and its notional machine, and (2) when using it for incremental program development. In other words: to make it teachable, as Wirth expressed it in his keynote at ITiCSE 2002 [Wirth 2002].

10.4 Programming methodology

The conceptual framework presented in chapter 7 enables new interpretations or improved understanding of well-known concepts such as stepwise refinement and correctness; from the perspective of incremental development, correctness can be conceived of as a quantitative property of a program.

10.4.1 Extension of STREAM

STREAM is a by-product of the conceptual framework of chapter 7. As emphasized in chapter 8, STREAM provides guidance with respect to refinement. Currently, guidance with respect to extension, the other main dimension of our incremental programming process, is provided through phrasing of exercises and assignments.

It is obvious to extend STREAM itself to address the principle of incremental program development. This work is in progress.

10.4.2 Theoretical foundation of conceptual framework

Currently, our description of incremental program development is based on a well-defined and consistent but yet informal foundation. It would be interesting and potentially rewarding to develop a formal foundation for incremental program development.

10.4.3 Extension of conceptual framework

The conceptual framework for incremental programming may be extended to characterize more aspects of programming processes. In particular, we are interested in *quantifying* properties of programs and activities related to programming.

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be.*
>
> — William Thomson, 1st Baron Kelvin, 1883

Traditionally, correctness is considered a qualitative property of programs. In section 7.1.5, we showed how to quantify the notion of correctness. It would be interesting to provide quantifications of other (so far conceived as purely) qualitative properties of programs.

Some solutions to a programming problem are more beautiful, elegant, and simple than others. We recognize a beautiful, elegant, and simple solution when we see one, and we may be able to vaguely characterize why we consider it to be so.

The principle of Ockham’s razor is often used when characterizing beautiful, elegant, and simple things. The principle states that the explanation of any phenomenon should make as few assumptions as possible, eliminating, or “shaving off”, those that make no difference in the observable predictions of the explanatory hypothesis or theory. The principle is often expressed in Latin as the “lex parsimoniae” (law of succinctness or parsimony): “entia non sunt multiplicanda praeter necessitatem”, which translates to: “entities should not be multiplied beyond necessity”. This is often paraphrased as “All things being equal, the simplest solution tends to be the best one.”

In the context of programming education, simplicity and understandability of example programs is essential; it is so partly for the purpose of reducing extraneous cognitive load but most importantly to serve as a model, an exemplary example —something to aspire for.

Quantification of simplicity and understandability, or at least partial quantification, is a challenge that would be extremely interesting to pursue. It is without doubt a hard challenge, but without pursuing it, we will never approach it. One way to approach understandability might be the following:

For any consistent mechanism, there are two key aspects: its *specification*, which describes what the mechanism does, and its *implementation*, which describes how it does it. In the following discussion, we ignore the requirements specification of a mechanism. We are interested in the understandability of a consistent version of a mechanism captured by its current specification and implementation.

One needs to understand the specification, and only the specification, to be able to use a mechanism. One needs to understand the implementation in order to grasp, explain, modify, or build a mechanism.

The dependency between the two is one way: understanding an implementation implies or requires understanding of the specification, but not vice versa.

Most of the time, we need to understand only the specification of mechanisms, and that is the primary purpose of the distinction between the two concepts. The secondary purpose is that *if* we need to understand the implementation (because we have to grasp, explain, modify, or build it), we can separate understanding of specification from understanding of implementation in two independent mental activities. Furthermore, we may —for a specific purpose, whatever that might be— be able to get away with only partial understanding of the implementation of a mechanism.

According to these observations, it makes perfectly good sense to measure understanding of specification and implementation independently of each other. Actually, it would be problematic if we cannot separate measurement of the two.

Consider the functions $u_{shallow}$ and u_{deep} denoting the cost of *shallow* and *deep understanding* of a mechanism. In terms of a third function, u, which denotes an unspecified cost of understanding of a specification or implementation, we may express $u_{shallow}$ and u_{deep} as follows (m is a mechanism, $m.s$ its specification, and $m.i$ its implementation):

$$u_{shallow}(m) = u(m.s)$$
$$u_{deep}(m) = u(m.s) + u(m.i)$$

The plus in the definition of u_{deep} expresses the separation of concern between understanding the specification of a mechanism and understanding its implementation.

Function u denotes an unspecified cost of understanding, to be refined later; for implementations it can be shallow, deep, or anything in between.

If the implementation of a mechanism $m.i$ is decomposed into part mechanisms (also called components), the cost of understanding these will show up in the expression of $u(m.i)$. Let us consider the special case where $m.i$ is decomposed into two components m_1 and m_2. In this situation, we request that $u(m.i)$ takes the form:

$$u(m.i) = e_{glue} + u(m_1) + u(m_2)$$

where e_{glue} represents the cost of understanding the “glue” of *m.i*, i.e. the part of *m.i* that glues together the use of m_1 and m_2 to implement *m*.

Each part mechanism m_1 and m_2 can in turn be understood at the shallow level or the deep level yielding four possible levels of understanding of *m*:

(1) $u(m.i) = e + u_{shallow}(m_1) + u_{shallow}(m_2)$
(2) $u(m.i) = e + u_{shallow}(m_1) + u_{deep}(m_2)$
(3) $u(m.i) = e + u_{deep}(m_1) + u_{shallow}(m_2)$
(4) $u(m.i) = e + u_{deep}(m_1) + u_{deep}(m_2)$

In general, any “horizontal” cut through the nodes of the decomposition tree of a mechanism represents a level of understanding. The cut need not be “straight”, i.e. the components or part-mechanisms need not all be broken down to the same level of understanding.

Figure 10-2 describes the level of understanding of a mechanism; the dashed line represents the *fringe* of understanding of the mechanism, i.e. a set of specifications of components constituting the implementation of the mechanism at a certain level of decomposition. Above the line, the understanding is deep because all implementation/glue is understood; at or below the line, understanding is shallow because only the specifications of the mechanisms constituting the fringe are understood.

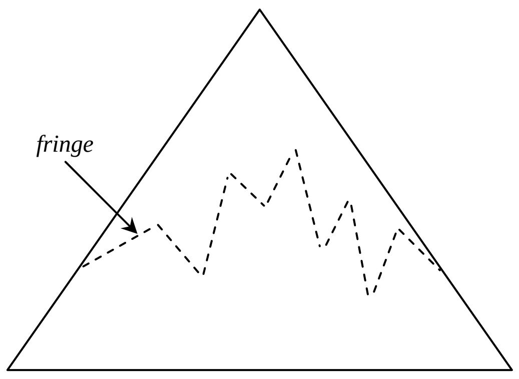


Figure 10-2: *The level of understanding of a mechanism*

The discussion above indicates how we might be able to exploit the conceptual framework of chapter 7 to approach the challenge of quantifying a notion of understandability of programs — a notion yet to be defined. Initial investigations along this line of thought have been carried out and are documented in chapter 22, where we discuss the development of a measurement framework for quality of example programs.

* * *

This concludes our discussion of potential future work enabled by this dissertation.

11 Conclusion

In this dissertation, we have addressed the grand challenge of programming education. The contributions of the dissertation encompass answers to five research questions related to three theses:

> **Thesis 1** (T_1): Revealing the programming process to novices eases and promotes the learning of programming.
>
> **Thesis 2** (T_2): Teaching skills as a supplement to knowledge promotes the learning of programming.
>
> **Thesis 3** (T_3): Anybody can learn to program.

The contributions of this dissertation encompass answers to five research questions derived from the three theses:

> **Research question 1**: What is the foundation in learning theory for programming education that supports T_1-T_3?

The first research question was addressed in chapter 3, where we provided an overview of elements of learning theory from cognitive science and educational psychology particularly relevant to our work. Based on a model of the human cognitive architecture, we provided a survey of cognitive load theory and cognitive skill acquisition that highlights the major achievements of the areas particularly relevant to introductory programming education.

> **Research question 2**: Does programming education research support T_1-T_3?

The second research question was addressed in chapter 4, where we presented a comprehensive overview of the overwhelming amount of programming education research by describing key conferences and publications relevant to the community and by capturing the essence of the research in programming education structured according to ten major research areas. The most striking indirect support for T_1 and T_2 in the huge body of research in programming education was the research on student understanding. The conclusion is clear and unambiguous: students struggle to learn programming, they are not very successful, and the major problem they experience is not syntax but how to put the pieces together. Teaching skills and exposing the programming process is all about teaching how to put the pieces together. There is no support of T_3 in the literature, on the contrary! The general theme of much research, and the specific results of some, is that (teaching) programming is hard and that students learn much less than we expect from them.

> **Research question 3**: Are there indicators of success for learning and performance in introductory programming?

The third research question was addressed in chapter 5, where we presented an overview of related work in the area of programming aptitude as well as three local studies. Our own studies have neither revealed new predictors of success nor been able to confirm the findings of others (except for a weak impact from math grade in high school). Our results may be regarded as negative, but we don't consider it that way. In light of T_3 the results are quite encouraging, and we speculate that our special flavour of an introductory programming course —a model-based approach to object-oriented programming with heavy emphasis on the programming process— has something to do with this. However, further research is required to conclude anything along these lines.

Research question 4: How does best-practice in modern software development relate to the research area of programming methodology?

The fourth research question was addressed in chapter 6 and 7. The short version of our findings is that best practice in modern software development and research in programming methodology relates very little to introductory programming education. Best practice is characterized by incremental program development whereas programming methodology research, as represented by the refinement calculus, adhere to the classical perspective of strict top-down refinement from abstract to concrete programs.

In chapter 6, we presented a perspective on the role of programming methodology in programming education and provided a brief overview of best-practice of modern software development. While programming methodology had some impact on education in the 1970s and 1980s, it vanished from the agenda when object-oriented programming emerged in the early 1990s.

Our study of the programming process of experts —which are confirmed by other studies— revealed that experts primarily progress according to a horizontal solve-a-simpler-problem-first-strategy similar to stepwise improvement. In the process, programmers apply opportunistic problem-solving, and the programming process is best characterised as an explorative activity of discovery and invention. Vertical approaches (strict top-down or bottom-up programming) are applied only in simple and trivial situations —the kind of situations that by definition never occur to novices.

In chapter 7, we exposed the fundamental difference between stepwise refinement and modern techniques of incremental program development and developed a conceptual framework for incremental program development that unifies the two. With specifications as the common denominator, we presented a conceptual framework for program extension that captures the essence of incremental program development and unifies classical stepwise refinement with best practices of software development such as refactoring and test-driven development. Our framework enables us to define correctness as a relative notion and to quantify the degree of correctness of a program.

A couple of simple examples were used to demonstrate how to put the conceptual framework into practice and to argue (once again) for the necessity of teaching novices about the programming process and to provide guidance about which increments to make and in which order to make them.

Research question 5: How can we educate novices in the skills of programming?

The fifth research question was addressed in chapter 8 and 9. In chapter 8, we presented our approach to programming education, which draws upon the results of cognitive load theory and offers an alternative to endless random walks. By providing guidance and scaffolding with respect to all dimensions involved in program development, we ensure that students exercise the important aspects of programming while keeping the cognitive load within the bounds where learning outcome is optimized. We described our primary means of providing guidance with respect to incremental development: through the structure of the teaching material (textbook, exercises and assignments, and videos) and by applying an apprentice-based approach to teaching.

Guidance with respect to refinement is provided through a carefully designed novice's process of object-oriented programming: STREAM, which we presented in chapter 8. The process, derived from the conceptual framework of chapter 7, represents a carefully scaled-down version of a full and rich software engineering process that is particularly suited for novices learning object-oriented programming.

In chapter 9, we described the rationale for the instructional design of an introductory programming course that incrementally approximates best-practice of modern software development; the instructional design was derived from the results of chapter 7 and 8. Based on a set of fundamental principles of programming education, we applied results of cognitive load theory, cognitive skill acquisition, and constructive alignment in particular to ensure an instructional design of an introductory programming course aimed at optimizing learning.

In chapter 10, we described a long list of potential future work derived from the PhD study.

* * *

We have not formally proved our research theses but we have made our case that revealing the programming process to novices and teaching skills as a supplement to knowledge promotes the learning of programming, and it is our experience that anybody —provided that they are motivated and that the body of knowledge is suitably structured— can learn the basic knowledge and skills of programming.

The grand challenge of programming education is not trivial, but there are strong indications that we are aiming in the right direction. Focus on the process of program development and the associated strategies, principles, patterns, and techniques is the missing link that we must provide in order to accomplish our mission of educating novices in the skills of programming.

Bibliography

[Abelson et al. 1995] Abelson, H., Bruce, K., Dam, A.v., Harvey, B., Tucker, A. and Wegner, P., "The first-course conundrum", *Commun ACM,* vol. 38, 6, pp. 116-117, 1995.

[ACM 2001] ACM, "Computing curricula 2001", *JERIC,* vol. 1, 3es, pp. 1, 2001.

[ACM 2005] ACM. "ACM Curricular Recommendations", http://www.acm.org/education/curricula.html, last updated: 2005, accessed: 2007.

[Adelson 1981] Adelson, B., "Problem solving and the development of abstract categories in programming languages", *Memory & Cognition,* vol. 9, 4, pp. 422-433, 1981.

[Adey et al. 1994] Adey, P. and Shayer, M., *Really raising standards. cognitive intervention and academic achievement,* London, Routledge, 1994.

[AgileAlliance 2007] AgileAlliance. "Agile Alliance", http://www.agilealliance.org/, last updated: 2007, accessed: 2007.

[Ala-Mutka 2005] Ala-Mutka, K.M., "A Survey of Automated Assessment Approaches for Programming Assignments", *Computer Science Education,* vol. 15, 2, pp. 83-102, 2005.

[Allen et al. 2002] Allen, E., Cartwright, R. and Stoler, B., "DrJava: a lightweight pedagogic environment for Java", *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education,* Cincinnati, Kentucky, pp. 137-141, 2002.

[Allsopp 2007] Allsopp, D.H. "Metacognitive Strategies", http://coe.jmu.edu/mathvidsr/metacognitive.htm, last updated: 2007, accessed: 2007.

[Alphonce et al. 2002] Alphonce, C. and Ventura, P., "Object orientation in CS1-CS2 by design", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 70-74, 2002.

[Alphonce et al. 2005] Alphonce, C. and Martin, B., "Green: a pedagogically customizable round-tripping UML class diagram Eclipse plug-in", *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange,* San Diego, California, pp. 115-119, 2005.

[Alphonce 2006] Alphonce, C. ""Killer Examples" for Design Patterns and Objects First Workshop", http://www.cse.buffalo.edu/~alphonce/KillerExamples/, last updated: 2006, accessed: 2006.

[Alphonce et al. 2007] Alphonce, C., Caspersen, M.E. and Decker, A., "Killer "Killer Examples" for Design Patterns", *SIGCSE '07: Proceedings of the 38th technical symposium on Computer science education,* Covington, Kentucky, USA, 2007.

[Anderson et al. 1989] Anderson, J.R., Conrad, F.G. and Corbett, A.T., "Skill acquisition and the LISP tutor", *Cognitive Science,* vol. 13, 4, pp. 467-505, 1989.

[Anderson 1993] Anderson, J.R., *Rules of the Mind,* Hillsdale, NJ, Erlbaum, 1993.

[Anderson et al. 1994] Anderson, J.R. and Fincham, J.M., "Acquisition of procedural skills from examples", *Journal of Experimental Psychology: Learning, Memory, and Cognition,* vol. 20, 6, pp. 1322-1340, 1994.

[Anderson et al. 2004] Anderson, R., Anderson, R., Simon, B., Wolfman, S.A., VanDeGrift, T. and Yasuhara, K., "Experiences with a tablet PC based lecture presentation system in computer science courses", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 56-60, 2004.

[Anjaneyulu 1994] Anjaneyulu, K.S.R., "Bug analysis of Pascal programs", *SIGPLAN Not.,* vol. 29, 4, pp. 15-22, 1994.

[Arthur 2006] Arthur, C. "How can I tell if I'll be any good as a programmer?", *The Guardian,* 2006.

[Astrachan 1991] Astrachan, O., "Pictures as invariants", *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education,* San Antonio, Texas, United States, pp. 112-118, 1991.

[Astrachan et al. 1995] Astrachan, O. and Reed, D., "AAA and CS 1: the applied apprenticeship approach to CS 1", *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education,* Nashville, Tennessee, United States, pp. 1-5, 1995.

[Astrachan et al. 1997] Astrachan, O., Smith, R. and Wilkes, J., "Application-based modules using apprentice learning for CS 2", *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education,* San Jose, California, United States, pp. 233-237, 1997.

[Astrachan et al. 1998] Astrachan, O., Mitchener, G., Berry, G. and Cox, L., "Design patterns: an essential component of CS curricula", *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 153-160, 1998.

[Atchison et al. 1968] Atchison, W.F., Conte, S.D., Hamblen, J.W., Hull, T.E., Keenan, T.A., Kehl, W.B., McCluskey, E.J., Navarro, S.O., Rheinboldt, W.C., Schweppe, E.J., Viavant, W. and Jr., D.M.Y., "Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science", *Commun ACM,* vol. 11, 3, pp. 151-197, 1968.

[Atkinson et al. 2000] Atkinson, R.K., Derry, S.J., Renkl, A. and Wortham, D., "Learning from Examples: Instructional Principles from the Worked Examples Research", *Review of Educational Research,* vol. 70, 2, pp. 181-214, 2000.

[Austin 1987] Austin, H.S., "Predictors of Pascal programming achievement for community college students", *SIGCSE Bull,* vol. 19, 1, pp. 161-164, 1987.

[Austing et al. 1979] Austing, R.H., Barnes, B.H., Bonnette, D.T., Engel, G.L. and Stokes, G., "Curriculum '78: recommendations for the undergraduate program in computer science — a report of the ACM curriculum committee on computer science", *Commun ACM,* vol. 22, 3, pp. 147-166, 1979.

[Back 1978] Back, R., "On the correctness of refinement steps in program development", Department of Computer Science, University of Helsinki, Helsinki, Finland, 1978.

[Bagert 1988] Bagert, D.J., "Should computer science examinations contain \“programming\” problems?", *SIGCSE '88: Proceedings of the nineteenth SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 288-292, 1988.

[Barker et al. 1983] Barker, R.J. and Unger, E.A., "A predictor for success in an introductory programming class based upon abstract reasoning development", *SIGCSE '83: Proceedings of the fourteenth SIGCSE technical symposium on Computer science education,* Orlando, Florida, United States, pp. 154-158, 1983.

[Barnes et al. 2006] Barnes, D.J. and Kölling, M., *Objects First with Java: A Practical Introduction Using BlueJ,* New York, Prentice Hall, pp. 520. 2006.

[Basili et al. 1974] Basili, V.R. and Turner, A.J., "Experiences with a simple structured programming language", *SIGCSE '74: Proceedings of the fourth SIGCSE technical symposium on Computer science education,* pp. 144-147, 1974.

[Beaubouef et al. 2005] Beaubouef, T. and Mason, J., "Why the high attrition rate for computer science students: some thoughts and observations", *SIGCSE Bull,* vol. 37, 2, pp. 103-106, 2005.

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

[Beck 2003] Beck, K., *Test-Driven Development: By Example,* Addison-Wesley, pp. 240. 2003.

[Becker 2006] Becker, K., "How much choice is too much?", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 78-82, 2006.

[Ben-Ari 2001] Ben-Ari, M., "Constructivism in Computer Science Education", *Journal of Computers in Mathematics and Science Teaching,* vol. 20, 1, pp. 45-73, 2001.

[Ben-Ari 2004] Ben-Ari, M., "Situated Learning in Computer Science Education", *Computer Science Education,* vol. 14, 2, pp. 85-100, 2004.

[Ben-Ari et al. 2004a] Ben-Ari, M., Berglund, A., Booth, S. and Holmboe, C., "What do we mean by theoretically sound research in computer science education?", *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* Leeds, United Kingdom, pp. 230-231, 2004.

[Ben-Ari et al. 2004b] Ben-Ari, M. and Sajaniemi, J., "Roles of variables as seen by CS educators", *SIGCSE Bull,* vol. 36, 3, pp. 52-56, 2004.

[Ben-Ari 2006a] Ben-Ari, M. "VN - Visualization of Nondeterminism", http://stwww.weizmann.ac.il/G-CS/BENARI/vn/index.html, last updated: 2006, accessed: 2007.

[Ben-Ari 2006b] Ben-Ari, M. "Tools for Teaching Concurrency with Spin", http://stwww.weizmann.ac.il/G-CS/BENARI/jspin/index.html, last updated: 2006, accessed: 2007.

[Ben-Ari 2006c] Ben-Ari, M., "McKinley's Amazon", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 75-77, 2006.

[Bennedsen et al. 2003] Bennedsen, J. and Caspersen, M., "Rationale for the Design of a Web-based Programming Course for Adults", *Procedings for the International Conference on Open and Online Learning (ICOOL 2003),* University of Mauritius, Mauritius, 2003.

[Bennedsen et al. 2004] Bennedsen, J. and Caspersen, M.E., "Programming in context: a model-first approach to CS1", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 477-481, 2004.

[Bennedsen et al. 2005a] Bennedsen, J. and Caspersen, M.E., "Revealing the programming process", *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, pp. 186-190, 2005.

[Bennedsen et al. 2005b] Bennedsen, J. and Caspersen, M.E., "An investigation of potential success factors for an introductory model-driven programming course", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 155-163, 2005.

[Bennedsen et al. 2006a] Bennedsen, J. and Caspersen, M.E., "Abstraction ability as an indicator of success for learning object-oriented programming?", *SIGCSE Bulletin,* vol. 38, 2, pp. 39-43, 2006.

[Bennedsen et al. 2006b] Bennedsen, J. and Caspersen, M., "Assessing Process and Product — A Practical Lab Exam for an Introductory Programming Course", *Procedings of the 36th Annual Frontiers in Education Conference,* San Diego, California, pp. M4E-16-M4E-21, 2006.

[Bennedsen et al. 2006c] Bennedsen, J., Caspersen, M.E. and Kölling, M. "SPoP — The Scandinavian Pedagogy of Programming Network", http://www.spop.dk/, last updated: 2006c, accessed: 2006.

[Bennedsen 2006e] Bennedsen, J., "The dissemination of pedagogical patterns", *Computer Science Education,* vol. 16, 2, pp. 119-136, 2006.

[Bennedsen et al. 2006f] Bennedsen, J. and Eriksen, O., "Categorizing Pedagogical patterns by teaching activities and Pedagogical values", *Computer Science Education,* vol. 16, 2, pp. 157-172, 2006.

[Bennedsen et al 2007a] Bennedsen, J., Caspersen, M.E. and Kölling, M., (Eds.) *Reflections on the Teaching of Programming.* Springer-Verlag, 2007.

[Bennedsen et al. 2007b] Bennedsen, J. and Caspersen, M.E., "Exposing the Programming Process". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

[Bennedsen et al. 2007c] Bennedsen, J. and Caspersen, M.E., "Model-Driven Programming". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

[Bennedsen et al. 2007d] Bennedsen, J. and Caspersen, M.E., "Failure Rates in Introductory Programming", *SIGCSE Bull,* vol. 39, 2, 2007.

[Bennedsen et al. 2007f] Bennedsen, J. and Schulte, C., "What does "Objects-First" Mean? An International Study of Teachers Perception of Objects-First", *Submitted to 12th Annual Conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland, 2007.

[Bergin et al. 1996] Bergin, J., Brodie, K., Patiño-Martínez, M., McNally, M., Naps, T., Rodger, S., Wilson, J., Goldweber, M., Khuri, S. and Jiménez-Peris, R., "An overview of visualization: its use and design (report of the working group in visualization)", *ITiCSE '96: Proceedings of the 1st conference on Integrating technology into computer science education,* Barcelona, Spain, pp. 192-200, 1996.

[Bergin et al. 2004] Bergin, J., Caristi, J., Dubinsky, Y., Hazzan, O. and Williams, L., "Teaching software development methods: the case of extreme programming", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 448-449, 2004.

[Bergin et al. 2005] Bergin, J., Stehlik, M., Robert, J. and Pattis, R., *Karel J Robot: A Gentle Introduction to theh Art of Object-Oriented Programming in Java,* Dream Songs Press, pp. 238. 2005.

[Bergin 2006a] Bergin, J. "Elementary Patterns", http://csis.pace.edu/~bergin/#elempat, last updated: 2006, accessed: 2007.

[Bergin 2006b] Bergin, J. "Pedagogical Patterns", http://csis.pace.edu/~bergin/#pedpat, last updated: 2006, accessed: 2007.

[Bergin 2007] Bergin, J., *Beyond Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java Volume 2,* Dream Songs Press, 2007.

[Bergin et al. 2005a] Bergin, S. and Reilly, R., "Programming: factors that influence success", *SIGCSE Bull,* vol. 37, 1, pp. 411-415, 2005.

[Bergin et al. 2005b] Bergin, S., Reilly, R. and Traynor, D., "Examining the role of self-regulated learning on introductory programming performance", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 81-86, 2005.

[Bergin et al. 2006] Bergin, S. and Reilly, R., "Predicting introductory programming performance: A multi-institutional multivariate study", *Computer Science Education,* vol. 16, 4, pp. 303-323, 2006.

[Berman et al. 1994] Berman, A.M., Decker, R., Nguyen, D.X., Reid, R.J. and Wallingford, E., "Using C++ in CS1/CS2", *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education,* Phoenix, Arizona, United States, pp. 383-384, 1994.

[Berman 1996] Berman, A.M., "On beyond OOP", *SIGPLAN Not.,* vol. 31, 4, pp. 1-3, 1996.

[Biggs 2003] Biggs, J.B., *Teaching for Quality Learning at University,* Open University Press, pp. 309. 2003.

[Blank et al. 2003] Blank, D., Meeden, L. and Kumar, D., "Python robotics: an environment for exploring robotics beyond LEGOs", *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education,* Reno, Navada, USA, pp. 317-321, 2003.

[Bloom et al. 1956] Bloom, B.S., Krathwohl, D.R. and Masia, B.B., *Taxonomy of educational objectives. the classification of educational goals. handbook I: Cognitive domain,* New York, Longmans, Green, 1956.

[Bloom 2007] Bloom, B. "Benjamin Bloom", http://en.wikipedia.org/wiki/Benjamin_Bloom, last updated: 2007, accessed: 2007.

[Booth 1997] Booth, S., "On Phenomenography, Learning, and Teaching", *Higher Education Research & Development,* vol. 16, 2, pp. 135-158, 1997.

[Börstler et al. 2003] Börstler, J. and Sharp, H., "Learning and Teaching Object Technology", *Computer Science Education,* vol. 13, 4, pp. 243-247, 2003.

[Börstler et al. 2007] Börstler, J., Caspersen, M.E. and Nordström, M., "Beauty and the Beast—Toward a Measurement Framework for Quality of Example Programs", *ITiCSE '07: Proceedings of the 12th international conference on Innovation and technology in computer science education,* Dundee, Scotland, 2007.

[Bovair et al. 1990] Bovair, S., Kieras, D.E. and Polson, P.G., "The Acquisition and Performance of Text-Editing Skill: A Cognitive Complexity Analysis", *Hum. -Comput. Interact.,* vol. 5, 1, pp. 1, 1990.

[Brabrand 2006] Brabrand, C. "Teaching Teaching & Understanding Understanding", Vol. DVD, 2006.

[Brilliant et al. 1996] Brilliant, S.S. and Wiseman, T.R., "The first programming paradigm and language dilemma", *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education,* Philadelphia, Pennsylvania, United States, pp. 338-342, 1996.

[Brown 2001] Brown, R.W., "Multi-choice versus descriptive examinations", Reno, NV, USA, pp. T3A-13-T3A-18, 2001.

[Brown 2006] Brown, B., "'The next line': Understanding programmers' work", *TeamEthno,* vol. 2, http://www.teamethno-online.org.uk/Issue2/, 2006.

[Bruce et al. 2001] Bruce, K.B., Danyluk, A.P. and Murtagh, T.P., "Event-driven programming is simple enough for CS1", *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education,* Canterbury, United Kingdom, pp. 1-4, 2001.

[Bruce et al. 2004] Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M. and Stoodley, I., "Ways of Experiencing the Act of Learnig to Program: A Phenomenographic Study of Introductory Programming Students at University", *Journal of Information Technology Education,* vol. 3, pp. 143-160, 2004.

[Bruce et al. 2005] Bruce, K., Danyluk, A. and Murtagh, T., *Java: An Eventful Approach,* Prentice-Hall, pp. 720. 2005.

[Bruner 1960] Bruner, J.S., *The Process of Education,* Cambridge, Mass., Harvard University Press, pp. 97. 1960.

[Bruner 2006] Bruner, J. "Jerome Bruner", http://en.wikipedia.org/wiki/Jerome_Bruner, last updated: 2006, accessed: 2007.

[Brünken et al. 2003] Brünken, R., Plass, J.L. and Leutner, D., "Direct Measurement of Cognitive Load in Multimedia Learning", *Educational Psychologist,* vol. 38, 1; 1, pp. 53-61, 2003.

[Brünken et al. 2004] Brünken, R., Plass, J.L. and Leutner, D., "Assessment of Cognitive Load in Multimedia Learning with Dual-Task Methodology: Auditory Load and Modality Effects", *Instructional Science,* vol. 32, 1-2, pp. 115-132, 2004.

[Brusilovsky et al. 2005a] Brusilovsky, P. and Sosnovsky, S., "Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK", *J. Educ. Resour. Comput.,* vol. 5, 3, pp. 6, 2005.

[Brusilovsky et al. 2005b] Brusilovsky, P. and Higgins, C., "Preface to the special issue on automated assessment of programming assignments", *J. Educ. Resour. Comput.,* vol. 5, 3, pp. 1, 2005.

[Buck et al. 2000] Buck, D. and Stucki, D.J., "Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development", *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education,* Austin, Texas, United States, pp. 75-79, 2000.

[Byrne et al. 2001] Byrne, P. and Lyons, G., "The effect of student attributes on success in programming", *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education,* Canterbury, United Kingdom, pp. 49-52, 2001.

[Cafolla 1988] Cafolla, R., "Piagetian Formal Operations and Other Cognitive Correlates of Achievement in Computer Programming.", *J. Educ. Technol. Syst.,* vol. 16, 1, pp. 45-55, 1988.

[Carroll 1994] Carroll, W.M., "Using worked examples as an instructional support in the algebra classroom", *J. Educ. Psychol.,* vol. 86, 3, pp. 360-367, 1994.

[Caspersen et al. 2000] Caspersen, M.E. and Christensen, H.B., "Here, there and everywhere — on the recurring use of turtle graphics in CS1", *ACSE '00: Proceedings of the Australasian conference on Computing education,* Melbourne, Australia, pp. 34-40, 2000.

[Caspersen et al. 2006a] Caspersen, M.E. and Kölling, M., "A novice's process of object-oriented programming", *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications,* Portland, Oregon, USA, pp. 892-900, 2006.

[Caspersen et al. 2007a] Caspersen, M.E., Bennedsen, J. and Larsen, K.D., "Mental Models and Programming Aptitude", *ITiCSE '07: The 12th annual conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland, 2007.

[Caspersen et al. 2007b] Caspersen, M.E. and Christensen, H.B., "CS1: Getting Started". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

[Catrambone 1998] Catrambone, R., "The subgoal learning model: Creating better examples so that students can solve novel problems", *J. Exp. Psychol. : Gen.,* vol. 127, 4, pp. 355-376, 1998.

[Chalk 2000] Chalk, P., "Webworlds—Web-Based Modeling Environments for Learning Software Engineering", *Computer Science Education,* vol. 10, 1, pp. 039-056, 2000.

[Chandler et al. 1991] Chandler, P. and Sweller, J., "Cognitive Load Theory and the Format of Instruction", *Cognition and Instruction,* vol. 8, 4, pp. 293-332, 1991.

[Chandler et al. 1992] Chandler, P. and Sweller, J., "The split attention effect as a factor in the design of instruction", *British Journal of Educational Psychology,* vol. 62, pp. 233-246, 1992.

[Chase et al. 1973] Chase, W.G. and Simon, H.A., "Perception in chess", *Cognitive Psychology,* vol. 4, 1, pp. 55-81, 1973.

[Chase et al. 1981] Chase, W.G. and Ericsson, K.A., "Skilled Memory". In *Cognitive Skills and Their Acquisition,* Hillsdale, NJ, Erlbaum, pp. 141-190, 1981.

[Chi et al. 1989] Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P. and Glaser, R., "Self-explanations: How students study and use examples in learning to solve problems", *Cognitive Science,* vol. 13, 2, pp. 145-182, 1989.

[Christensen et al. 2002] Christensen, H.B. and Caspersen, M.E., "Frameworks in CS1: a different way of introducing event-driven programming", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 75-79, 2002.

[Christensen 2004] Christensen, H.B., "Frameworks: putting design patterns into perspective", *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* Leeds, United Kingdom, pp. 142-145, 2004.

[Christensen et al. 2007] Christensen, H.B. and Caspersen, M.E., "Frameworks and their Role in Teaching". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

[Clancy et al. 1999] Clancy, M.J. and Linn, M.C., "Patterns and pedagogy", *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education,* New Orleans, Louisiana, United States, pp. 37-42, 1999.

[Clancy et al. 2001] Clancy, M., Stasko, J., Guzdial, M., Fincher, S. and Dale, N., "Models and Areas for CS Education Research", *Computer Science Education,* vol. 11, 4, pp. 323-341, 2001.

[Clancy 2004] Clancy, M.J., "Misconceptions and Attitudes that Interfere with Learning to Program". In *Computer Science Education Research,* Taylor & Francis, pp. 85-100, 2004.

[Clark et al. 2006] Clark, R., Nguyen, F. and Sweller, J., *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load,* John Wiley & Sons, pp. 390. 2006.

[Cockburn 2002] Cockburn, A., *Agile software development,* Boston, Addison-Wesley, 2002.

[Collins et al. 1989] Collins, A., Brown, J.S. and Newman, S.E., "Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics". In *Knowing, learning and instruction: Essays in honour of Robert Glaser,* Hillsdale, NJ, Erlbaum, 1989.

[Collins et al. 1991] Collins, A.M., Brown, J.S. and Holum, A., "Cognitive apprenticeship: Making thinking visible", *American Educator,* vol. 15, 3, pp. 6-11, 38-46, 1991.

[Comer et al. 1989] Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J. and Young, P.R., "Computing as a discipline", *Commun ACM,* vol. 32, 1, pp. 9-23, 1989.

[Cooper et al. 1987] Cooper, G. and Sweller, J., "Effects of schema acquisition and rule automation on mathematical problem-solving transfer", *J. Educ. Psychol.,* vol. 79, 4, pp. 347-362, 1987.

[Cooper et al. 2003] Cooper, S., Dann, W. and Pausch, R., "Teaching objects-first in introductory computer science", *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education,* Reno, Navada, USA, pp. 191-195, 2003.

[Culwin 1997] Culwin, F., "Java in the C.S. curriculum (seminar)", *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education,* San Jose, California, United States, pp. 392, 1997.

[Cuny et al. 2002] Cuny, J. and Aspray, W., "Recruitment and retention of women graduate students in computer science and engineering: results of a workshop organized by the computing research association", *SIGCSE Bull,* vol. 34, 2, pp. 168-174, 2002.

[CUSE 1997] CUSE, *Science Teaching Reconsidered: A Handbook,* National Academy Press, pp. 88. 1997.

[Dahl et al. 1966] Dahl, O. and Nygaard, K., "SIMULA: an ALGOL-based simulation language", *Commun ACM,* vol. 9, 9, pp. 671-678, 1966.

[Dahl et al. 1972] Dahl, O.-., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming,* London, Academic Press, pp. 220. 1972.

[Dahlbom et al. 1997] Dahlbom, B. and Mathiassen, L., "The future of our profession", *Commun ACM,* vol. 40, 6, pp. 80-89, 1997.

[Dale 2002] Dale, N., "Increasing interest in CS ed research", *SIGCSE Bull,* vol. 34, 4, pp. 16-17, 2002.

[Dale 2004] Dale, N. "Course Content Survey Results (publisher's list group)", http://www.cs.utexas.edu/users/ndale/ContentResults2.html, last updated: 2004, accessed: 2006.

[Dale 2005] Dale, N., "Content and emphasis in CS1", *SIGCSE Bull,* vol. 37, 4, pp. 69-73, 2005.

[Dale 2006] Dale, N.B., "Most difficult topics in CS1: results of an online survey of educators", *SIGCSE Bull,* vol. 38, 2, pp. 49-53, 2006.

[Daly et al. 2004] Daly, C. and Waldron, J., "Assessing the assessment of programming ability", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 210-213, 2004.

[Daly et al. 2005] Daly, C. and Horgan, J., "Patterns of plagiarism", *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, pp. 383-387, 2005.

[Daniels et al. 1999] Daniels, M., Berglund, A. and Petre, M., "Reflections on International Projects in Undergraduate CS Education", *Computer Science Education,* vol. 9, 3, pp. 256-267, 1999.

[Danielson et al. 1975] Danielson, R.L. and Nievergelt, J., "An automatic tutor for introductory programming students", *SIGCSE '75: Proceedings of the fifth SIGCSE technical symposium on Computer science education,* pp. 47-50, 1975.

[Davey et al. 2002] Davey, B.A. and Priestley, H.A., *Introduction to Lattices and Order,* Cambridge University Press, pp. 298. 2002.

[Dean et al 2004] Dean, C.N. and Boute, R.T., (Eds.) *Teaching Formal Methods.* , vol. 3294, Springer-Verlag, 2004. pp. 249.

[Decker et al. 1992] Decker, R. and Hirshfield, S., "A case for, and an instance of, objects in CS1", *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum),* Vancouver, British Columbia, Canada, pp. 309-312, 1992.

[Decker et al. 1993] Decker, R. and Hirshfield, S., "Top-down teaching: object-oriented programming in CS 1", *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education,* Indianapolis, Indiana, United States, pp. 270-273, 1993.

[Decker et al. 1994] Decker, R. and Hirshfield, S., "The top 10 reasons why object-oriented programming can't be taught in CS 1", *SIGCSE Bull,* vol. 26, 1, pp. 51-55, 1994.

[Deek 1999] Deek, F.P., "The Software Process: A Parallel Approach through Problem Solving and Program Development", *Computer Science Education,* vol. 9, 1, pp. 43-70, 1999.

[Dehnadi et al. 2006a] Dehnadi, S. and Bornat, R. "The camel has two humps", 2006.

[Dehnadi 2006b] Dehnadi, S., "Testing programming aptitude", *Procedings of the 18th Annual Workshop of the Psychology of Programming Interest Group,* Brighton, UK, pp. 22-37, 2006.

[Denning 1975] Denning, P.J., "Two misconceptions about structured programming", *ACM 75: Proceedings of the 1975 annual conference,* pp. 214-215, 1975.

[Denning et al 1997] Denning, P.J. and Metcalfe, R.M., (Eds.) *Beyond Calculation: The Next Fifty Years of Computing.* Springer-Verlag, 1997. pp. 313.

[Denning 2001] Denning, P.J., "The profession of IT: crossing the chasm", *Commun ACM,* vol. 44, 4, pp. 21-25, 2001.

[Denning 2002] Denning, P.J., "Flatlined", *Commun ACM,* vol. 45, 6, pp. 15-19, 2002.

[Denning 2003] Denning, P.J., "Great principles of computing", *Commun ACM,* vol. 46, 11, pp. 15-20, 2003.

[Denning et al. 2004] Denning, P., Johnson, C., Utting, I., Cassel, L. and Clark, M. "Position Papers for Conference on Grand Challenges in Computing Education", http://www.cis.strath.ac.uk/external/educ_grand_challenges/programme.html, last updated: 2004, accessed: 2006.

[Denning 2004a] Denning, P. "Programming as Practice, Grand Challenges in Education", 2004.

[Denning 2004b] Denning, P.J., "The field of programmers myth", *Commun ACM,* vol. 47, 7, pp. 15-20, 2004.

[Denning et al. 2005] Denning, P.J. and McGettrick, A., "Recentering computer science", *Commun ACM,* vol. 48, 11, pp. 15-19, 2005.

[Denning et al. 2006] Denning, T., Griswold, W.G., Simon, B. and Wilkerson, M., "Multimodal communication in the classroom: what does it mean for us?", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 219-223, 2006.

[Dick et al. 2002] Dick, M., Sheard, J., Bareiss, C., Carter, J., Joyce, D., Harding, T. and Laxer, C., "Addressing student cheating: definitions and solutions", *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 172-184, 2002.

[Dijkstra 1969] Dijkstra, E.W., "Notes on structured programming", Tech. Rep. EWD 249, 1969.

[Dijkstra 1975] Dijkstra, E.W., "Guarded commands, nondeterminacy and formal derivation of programs", *Commun ACM,* vol. 18, 8, pp. 453-457, 1975.

[Dijkstra 1976] Dijkstra, E.W., *A Discipline of Programming,* Englewood Cliffs, New Jersey, Prentice-Hall, pp. 217. 1976.

[Dijkstra 1989] Dijkstra, E.W., "On the Cruelty of Really Teaching Computing Science", *Communications of the ACM,* vol. 32, 12, pp. 1398-1404, 1989.

[Dijkstra et al. 1989] Dijkstra, E.W., Parnas, D.L., Scherlis, W., van Emden, M.H., Cohen, J., Hamming, R., Karp, R.M. and Winograd, T., "A debate on teaching computing science", *Commun ACM,* vol. 32, 12, pp. 1397-1414, 1989.

[Douce et al. 2005] Douce, C., Livingstone, D. and Orwell, J., "Automatic test-based assessment of programming: A review", *J. Educ. Resour. Comput.,* vol. 5, 3, pp. 4, 2005.

[Dromey 1982] Dromey, R.G., *How to Solve it by Computer,* Prentice-Hall International Series in Computer Science, pp. 442. 1982.

[du Boulay 1989a] du Boulay, B., "Some difficulties of learning to program.". In *Studying the novice programmer,* Hillsdale, NJ, Lawrence Erlbaum, pp. 57-73, 1989.

[du Boulay et al. 1989b] du Boulay, B., O'Shea, T. and Monk, J., "The black box in-side the glass box: presenting computing concepts to novices.". In *Studying the novice programmer,* Hillsdale, NJ, Lawrence Erlbaum, 1989.

[Dupras et al. 1984] Dupras, M., LeMay, F. and Mili, A., "Some thoughts on teaching first year programming", *SIGSCE '84: Proceedings of the fifteenth SIGCSE technical symposium on Computer science education,* pp. 148-153, 1984.

[East et al. 1996] East, J.P., Thomas, S.R., Wallingford, E., Beck, W. and Drake, J., "Pattern-based Programming Instruction", Washinghton DC, 1996.

[Ebel et al. 2006] Ebel, G. and Ben-Ari, M., "Affective effects of program visualization", *ICER '06: Proceedings of the 2006 international workshop on Computing education research,* Canterbury, United Kingdom, pp. 1-5, 2006.

[Eckerdal et al. 2005a] Eckerdal, A. and Thuné, M., "Novice Java programmers' conceptions of "object" and "class", and variation theory", *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* Caparica, Portugal, pp. 89-93, 2005.

[Eckerdal et al. 2005b] Eckerdal, A., Thuné, M. and Berglund, A., "What does it take to learn 'programming thinking'?", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 135-142, 2005.

[Eckerdal et al. 2006] Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M. and Zander, C., "Can graduating students design software systems?", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 403-407, 2006.

[Eckstein 2001] Eckstein, J., "Pedagogical patterns: Capturing best practice in teaching object technology", *Software Focus,* vol. 2, 1, pp. 9-12, 2001.

[Edwards et al. 2000] Edwards, H.M., Thompson, B.J., Halstead-Nussloch, R., Arnow, D. and Oliver, D., "Report on the CSEET '99 Workshop: “Establishing a Distance Education Program”", *Computer Science Education,* vol. 10, 1, pp. 057-074, 2000.

[Edwards 2003a] Edwards, S.H., "Rethinking computer science education from a test-first perspective", *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* Anaheim, CA, USA, pp. 148-155, 2003.

[Edwards 2003b] Edwards, S.H., "Improving student performance by evaluating how well students test their own programs", *J. Educ. Resour. Comput.,* vol. 3, 3, pp. 1, 2003.

[Edwards 2004] Edwards, S.H., "Using software testing to move students from trial-and-error to reflection-in-action", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 26-30, 2004.

[Evans et al. 1989] Evans, G.E. and Simkin, M.G., "What best predicts computer proficiency?", *Communication of the ACM,* vol. 32, 11, pp. 1322-1327, 1989.

[Fagin et al. 2003] Fagin, B. and Merkle, L., "Measuring the effectiveness of robots in teaching computer science", *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education,* Reno, Navada, USA, pp. 307-311, 2003.

[Fincher et al. 2002] Fincher, S. and Utting, I., "Pedagogical patterns: their place in the genre", *SIGCSE Bull,* vol. 34, 3, pp. 199-202, 2002.

[Fincher et al. 2004] Fincher, S. and Petre, M., *Computer science education research,* London, Routledge Falmer, 2004.

[Fincher 2006] Fincher, S. "Special issue on CSE Pedagogic patterns", *Computer Science Education,* Vol. 16, pp. 75-75, 2006.

[Findler et al. 2002] Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. and Felleisen, M., "DrScheme: a programming environment for Scheme", *J. Funct. Program.,* vol. 12, 2, pp. 159-182, 2002.

[Fowler 1999] Fowler, M., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, pp. 464. 1999.

[Fowler et al. 2000] Fowler, M. and Scott, K., *UML Distilled: A Brief Guide to the Standard Object Modeling Language,* Addison-Wesley, pp. 185. 2000.

[Fowler 2003] Fowler, M., "When to Make a Type", *IEEE Software,* vol. 20, 1, pp. 12-13, 2003.

[Fox 1997] Fox, J., *Applied regression analysis, linear models, and related models,* Sage Publications, pp. 597. 1997.

[Frandsen et al. 2006] Frandsen, G.S. and Schwartzbach, M.I., "A singular choice for multiple choice", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 34-38, 2006.

[Gamma 1995] Gamma, E., *Design patterns. elements of reusable object-oriented software,* Reading, Mass., Addison-Wesley, 1995.

[Gantenbein 1989] Gantenbein, R.E., "Programming as process: a “novel” approach to teaching programming", *SIGCSE '89: Proceedings of the twentieth SIGCSE technical symposium on Computer science education,* Louisville, Kentucky, United States, pp. 22-26, 1989.

[Gardner 1983] Gardner, H., *Frames of the Mind: The Theory of the Multiple Intelligence,* BasicBooks, pp. 432. 1983.

[GCC 2004] GCC. "Grand Challenges for Computing Education and Research", http://www.cs.ncl.ac.uk/research/events/conferences/2004/GCC04/index.htm, last updated: 2004, accessed: 2006.

[Gelfand et al. 1998] Gelfand, N., Goodrich, M.T. and Tamassia, R., "Teaching data structure design patterns", *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 331-335, 1998.

[Gerjets et al. 2004] Gerjets, P., Scheiter, K. and Catrambone, R., "Designing Instructional Examples to Reduce Intrinsic Cognitive Load: Molar versus Modular Presentation of Solution Procedures", *Instructional Science,* vol. 32, 1-2, pp. 33-58, 2004.

[Gersting 2000] Gersting, J.L., "Computer Science Distance Education Experience in Hawaii", *Computer Science Education,* vol. 10, 1, pp. 095-106, 2000.

[Gibbon et al. 1996] Gibbon, C.A. and Higgins, C.A., "Towards a Learner-Centred Approach to Teaching Object-Oriented Design", *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference,* pp. 110, 1996.

[Gibbs 2000] Gibbs, D.C., "The effect of a constructivist learning environment for field-dependent/independent students on achievement in introductory computer programming", *SIGCSE Bull,* vol. 32, 1, pp. 207-211, 2000.

[Gick et al. 1983] Gick, M.L. and Holyoak, K.J., "Schema induction and analogical transfer", *Cognitive Psychology,* vol. 15, 1, pp. 1-38, 1983.

[Gilmore 1990] Gilmore, D., "Methodological issues in the study ofprogramming". In *Psychology of Programming,* Academic Press, pp. 83-98, 1990.

[Goldberg et al. 1983] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation,* Boston, MA, Addison-Wesley, 1983.

[Goold et al. 2000] Goold, A. and Rimmer, R., "Factors affecting performance in first-year computing", *SIGCSE Bull,* vol. 32, 2, pp. 39-43, 2000.

[Gosling et al. 2005] Gosling, J., Joy, B., Steele, G. and Bracha, G., *The Java Language Specification,* Addison-Wesley, pp. 649. 2005.

[Gries 1974] Gries, D., "What should we teach in an introductory programming course?", *SIGCSE '74: Proceedings of the fourth SIGCSE technical symposium on Computer science education,* pp. 81-89, 1974.

[Gries 1978] Gries, D., (Eds.) *Programming Methodology.* New York, Springer-Verlag, 1978. pp. 437.

[Gries 1981] Gries, D., *The Science of Programming,* New York, Springer-Verlag, pp. 366. 1981.

[Gries 2002] Gries, D., "Where is programming methodology these days?", *SIGCSE Bull,* vol. 34, 4, pp. 5-7, 2002.

[Gries et al. 2002] Gries, P. and Gries, D., "Frames and folders: a teachable memory model for Java", *J. Comput. Small Coll.,* vol. 17, 6, pp. 182-196, 2002.

[Gries 2006] Gries, D., "What Have We Not Learned about Teaching Programming?", *IEEE Computer,* vol. 39, 10, pp. 81-82, 2006.

[Gross et al. 2005] Gross, P. and Powers, K., "Evaluating assessments of novice programming environments", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 99-110, 2005.

[Guzdial 2004] Guzdial, M., "Programming Environments for Novices". In *Computer science education research,* London, Routledge Falmer, pp. 127-154, 2004.

[Guzdial 2006] Guzdial, M. "Failure rates", 2006.

[Haaster et al. 2004] Haaster, K.V. and Hagan, D., "Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool", *Information Science + Information Technology Education Joint Conference,* Rockhampton, Queensland, Australia, pp. 455-470, 2004.

[Hadjerrouit 1999] Hadjerrouit, S., "A constructivist approach to object-oriented design and programming", *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education,* Cracow, Poland, pp. 171-174, 1999.

[Hadjerrouit 2005] Hadjerrouit, S., "Constructivism as guiding philosophy for software engineering education", *SIGCSE Bull,* vol. 37, 4, pp. 45-49, 2005.

[Hagan et al. 2000] Hagan, D. and Markham, S., "Does it help to have some programming experience before beginning a computing degree program?", *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education,* Helsinki, Finland, pp. 25-28, 2000.

[Hansen et al. 2002] Hansen, K.M. and Ratzer, A.V., "Tool support for collaborative teaching and learning of object-oriented modeling", *ITiCSE '02: Proceedings of the 7th annual* *conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 146-150, 2002.

[Hansen et al. 2004] Hansen, S. and Fossum, T., "Events not equal to GUIs", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 378-381, 2004.

[Harlan et al. 2001] Harlan, R.M., Levine, D.B. and McClarigan, S., "The Khepera robot and the kRobot class: a platform for introducing robotics in the undergraduate curriculum", *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education,* Charlotte, North Carolina, United States, pp. 105-109, 2001.

[Hazzan et al. 2006] Hazzan, O., Dubinsky, Y., Eidelman, L., Sakhnini, V. and Teif, M., "Qualitative research in computer science education", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 408-412, 2006.

[Heathcote et al. 2000] Heathcote, A., Brown, S. and Mewhort, D.J., "The power law repealed: the case for an exponential law of practice", *Psychonomic Bulletin & Review,* vol. 7, 2, pp. 185-207, 2000.

[Hegna et al. 2006] Hegna, H. and Groven, A., "A study of objects-first with BlueJ in a non-computer science context". In *Comprehensive Object-Oriented Learning: The Learnes's Perspective,* Informing Science Press, pp. 79-110, 2006.

[Henriksen et al. 2004] Henriksen, P. and Kölling, M., "greenfoot: combining object visualisation with interaction", *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications,* Vancouver, BC, CANADA, pp. 73-82, 2004.

[Hesketh et al. 1989] Hesketh, B., Andrews, S. and Chandler, P., "Opinion--Training for Transferable Skills: The Role of Examples and Schema", *Education and Training Technology International,* vol. 26, 2, pp. 105-156, 1989.

[Holden et al. 2003] Holden, E. and Weeden, E., "The impact of prior experience in an information technology programming course sequence", *CITC4 '03: Proceedings of the 4th conference on Information technology curriculum,* Lafayette, Indiana, USA, pp. 41-46, 2003.

[Holland et al. 1997] Holland, S., Griffiths, R. and Woodman, M., "Avoiding object misconceptions", *SIGCSE Bull,* vol. 29, 1, pp. 131-134, 1997.

[Holt 1994] Holt, R.C., "Introducing undergraduates to object orientation using the Turing language", *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education,* Phoenix, Arizona, United States, pp. 324-328, 1994.

[Hostetler 1983] Hostetler, T.R., "Predicting student success in an introductory programming course", *SIGCSE Bull,* vol. 15, 3, pp. 40-43, 1983.

[Hu 2005] Hu, C., "Dataless objects considered harmful", *Commun ACM,* vol. 48, 2, pp. 99-101, 2005.

[Hundhausen et al. 2002] Hundhausen, C.D., Douglas, S.A. and Stasko, J.T., "A Meta-Study of Algorithm Visualization Effectiveness", *Journal of Visual Languages & Computing,* vol. 13, 3, pp. 259-290, 2002.

[Hundhausen et al. 2006] Hundhausen, C.D., Brown, J.L., Farley, S. and Skarpas, D., "A methodology for analyzing the temporal evolution of novice programs based on semantic components", *ICER '06: Proceedings of the 2006 international workshop on Computing education research,* Canterbury, United Kingdom, pp. 59-71, 2006.

[Hyman et al. 1965] Hyman, R. and Anderson, B., "Solving Problems", *International Science and Technology,* pp. 36-41, 1965.

[Ibbett et al. 2006] Ibbett, R.N., Carballo, J.C.D.y. and Dolman, D.A.W., "Computer architecture simulation models", *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education,* Bologna, Italy, pp. 353-353, 2006.

[IFIP 2006] IFIP. "IFIP WG 2.3: Programming Methodology", http://www.ifip.org/bulletin/bulltcs/memtc02.htm, last updated: 2006, accessed: 2007.

[Imberman et al. 2005] Imberman, S.P. and Klibaner, R., "A robotics lab for CS1", *J. Comput. Small Coll.,* vol. 21, 2, pp. 131-137, 2005.

[Ingalls 1978] Ingalls, D.H.H., "The Smalltalk-76 programming system design and implementation", *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages,* Tucson, Arizona, pp. 9-16, 1978.

[Inhelder et al. 1958] Inhelder, B. and Piaget, J., *The growth of logical thinking from childhood to adolescence. an essay on the construction of formal operational structures,* , vol. 7. print., New York, Basic Books, 1958.

[Jadud 2006] Jadud, M.C., "Methods and tools for exploring novice compilation behaviour", *ICER '06: Proceedings of the 2006 international workshop on Computing education research,* Canterbury, United Kingdom, pp. 73-84, 2006.

[Cross et al. 2006] James H. Cross, I. and T. Dean Hendrix, "jGRASP: a lightweight IDE with dynamic object viewers for CS1 and CS2", *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education,* Bologna, Italy, pp. 356-356, 2006.

[Janzen et al. 2006] Janzen, D.S. and Saiedian, H., "Test-driven learning: intrinsic integration of testing into the CS/SE curriculum", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 254-258, 2006.

[Jargon 2003] Jargon. "The Jargon File", http://catb.org/jargon/, last updated: 2003, accessed: 2006.

[Jeffries et al. 1981] Jeffries, R., Turner, A.A., Polson, P.G. and Atwood, M.E., "The Processes Involved in Designing Software". In *Cognitive Skills and Their Acquisition,* Hillsdale, NJ, Erlbaum, pp. 255-284, 1981.

[Jones 2004] Jones, C.G., "Test-driven development goes to school", *J. Comput. Small Coll.,* vol. 20, 1, pp. 220-231, 2004.

[Kalyuga et al. 2003] Kalyuga, S., Ayres, P., Chandler, P. and Sweller, J., "The Expertise Reversal Effect", *Educational Psychologist,* vol. 38, 1; 1, pp. 23-31, 2003.

[Karavirta et al. 2006] Karavirta, V., Korhonen, A. and Malmi, L., "On the use of resubmissions in automatic assessment systems", *Computer Science Education,* vol. 16, 3, pp. 229-240, 2006.

[Kirschner 2002] Kirschner, P.A., "Cognitive load theory: implications of cognitive load theory on the design of learning", *Learning and Instruction,* vol. 12, 1, pp. 1-10, 2002.

[Knudsen et al. 1988] Knudsen, J.L. and Madsen, O.L., "Teaching Object-Oriented Programming is more than teaching Object-Oriented Programming Languages", *ECOOP '88 European Conference on Object-Oriented Programming,* Oslo, Norway, pp. 21-40, 1988.

[Knuth 1974] Knuth, D.E., "Structured Programming with go to Statements", *ACM Comput. Surv.,* vol. 6, 4, pp. 261-301, 1974.

[Koile et al. 2006] Koile, K. and Singer, D., "Improving learning in CS1 via tablet-PC-based in-class assessment", *ICER '06: Proceedings of the 2006 international workshop on Computing education research,* Canterbury, United Kingdom, pp. 119-126, 2006.

[Kolb et al. 1975] Kolb, D.A. and Fry, R., "Toward an applied theory of experiential learning". In *Theories of Group Processes,* John Wiley & Sons, 1975.

[Koli 2006] Koli. "Koli Calling", http://cs.joensuu.fi/kolistelut/, last updated: 2006, accessed: 2006.

[Kölling et al. 1995] Kölling, M., Koch, B. and Rosenberg, J., "Requirements for a first year object-oriented teaching language", *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education,* Nashville, Tennessee, United States, pp. 173-177, 1995.

[Kölling et al. 1996a] Kölling, M. and Rosenberg, J., "Blue—a language for teaching object-oriented programming", *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education,* Philadelphia, Pennsylvania, United States, pp. 190-194, 1996.

[Kölling et al. 1996b] Kölling, M. and Rosenberg, J., "An object-oriented program development environment for the first programming course", *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education,* Philadelphia, Pennsylvania, United States, pp. 83-87, 1996.

[Kölling 2003a] Kölling, M. "Unit Testing in BlueJ", http://www.bluej.org/tutorial/testing-tutorial.pdf, last updated: 2003, accessed: 2007.

[Kölling 2003b] Kölling, M., "The Curse of Hello World", *Workshop on Learning and Teaching Object-orientation – Scandinavian Perspectives,* Oslo, 2003.

[Kölling et al. 2003] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., "The BlueJ system and its pedagogy", *Journal of Computer Science Education,* vol. 13, 4, pp. 249-268, 2003.

[Kölling et al. 2004] Kölling, M. and Barnes, D.J., "Enhancing apprentice-based learning of Java", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 286-290, 2004.

[Kölling et al. 2005] Kölling, M. and Henriksen, P., "Game programming in introductory courses with direct state manipulation", *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* Caparica, Portugal, pp. 59-63, 2005.

[Konvalina et al. 1983] Konvalina, J., Wileman, S.A. and Stephens, L.J., "Math proficiency: a key to success for computer science students", *Commun ACM,* vol. 26, 5, pp. 377-382, 1983.

[Korhonen et al. 2002] Korhonen, A., Malmi, L., Myllyselkä, P. and Scheinin, P., "Does it make a difference if students exercise on the web or in the classroom?", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 121-124, 2002.

[Kristensen et al. 2007] Kristensen, B.B., Madsen, O.L. and Møller-Petersen, B., "The When, Why and Why Not of the BETA Programming Language", San Diego, California, 2007.

[Kumar 2005a] Kumar, A.N., "Results from the evaluation of the effectiveness of an online tutor on expression evaluation", *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, pp. 216-220, 2005.

[Kumar 2005b] Kumar, A.N., "Generation of problems, answers, grade, and feedback—case study of a fully automated tutor", *JERIC,* vol. 5, 3, pp. 1-25, 2005.

[Kumar 2007] Kumar, A.N. "Problets", http://www.problets.org/, last updated: 2007, accessed: 2007.

[Kurtz 1980] Kurtz, B.L., "Investigating the relationship between the development of abstract reasoning and performance in an introductory programming class", *SIGCSE '80: Proceedings of the eleventh SIGCSE technical symposium on Computer science education,* Kansas City, Missouri, United States, pp. 110-117, 1980.

[Lahtinen et al. 2005] Lahtinen, E., Ala-Mutka, K. and Järvinen, H., "A study of the difficulties of novice programmers", *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* Caparica, Portugal, pp. 14-18, 2005.

[Lancaster et al. 2004] Lancaster, T. and Culwin, F., "A Comparison of Source Code Plagiarism Detection Engines", *Computer Science Education,* vol. 14, 2, pp. 101-117, 2004.

[Last et al. 2002] Last, M.Z., Daniels, M., Hause, M.L. and Woodroffe, M.R., "Learning from students: continuous improvement in international collaboration", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 136-140, 2002.

[Lave et al. 1991] Lave, J. and Wenger, E., *Situated learning. Legitimate peripheral participation,* Cambridge, UK, Cambridge University Press, 1991.

[Lawhead et al. 2002] Lawhead, P.B., Duncan, M.E., Bland, C.G., Goldweber, M., Schep, M., Barnes, D.J. and Hollingsworth, R.G., "A road map for teaching introductory programming using LEGO mindstorms robots", *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 191-201, 2002.

[Lee 2007] Lee, M. "CS1: Weaker students avoid coding like the plague!", http://listserv.acm.org/scripts/wa.exe?A2=ind0701d&L=sigcse-members&F=&S=&P=1423, last updated: 2007, accessed: 2007.

[Leeper et al. 1982] Leeper, R.R. and Silver, J.L., "Predicting success in a first programming course", *SIGCSE '82: Proceedings of the thirteenth SIGCSE technical symposium on Computer science education,* Indianapolis, Indiana, United States, pp. 147-150, 1982.

[LeFevre et al. 1986] LeFevre, J. and Dixon, P., "Do Written Instructions Need Examples?", *Cognition & Instruction,* vol. 3, 1; 1, pp. 1, 1986.

[Lindholm 2005] Lindholm, M., "Development of object-understanding among students in the humanities", *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* Caparica, Portugal, pp. 382-382, 2005.

[Lindholm 2007] Lindholm, M., "Conceptions of Object-Oriented Terms: A study in progress", Salford, 2007.

[Linn et al. 1985] Linn, M.C. and Dalbey, J., "Cognitive consequences of Programming Instruction: Instruction, Access, and Ability", *Educational Psychologist,* vol. 20, 4; 4, pp. 191, 1985.

[Linn et al. 1992] Linn, M.C. and Clancy, M.J., "The case for case studies of programming problems", *Commun ACM,* vol. 35, 3, pp. 121-132, 1992.

[Lister 2003] Lister, R., "A research manifesto, and the relevance of phenomenography", *SIGCSE Bull,* vol. 35, 2, pp. 15-16, 2003.

[Lister et al. 2003] Lister, R. and Leaney, J., "Introductory programming, criterion-referencing, and bloom", *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education,* Reno, Navada, USA, pp. 143-147, 2003.

[Lister et al. 2004] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L., "A multi-national study of reading and tracing skills in novice programmers", *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education,* Leeds, United Kingdom, pp. 119-150, 2004.

[Lister 2005a] Lister, R., "Grand challenges", *SIGCSE Bull,* vol. 37, 2, pp. 14-15, 2005.

[Lister et al. 2006] Lister, R., Simon, B., Thompson, E., Whalley, J.L. and Prasad, C., "Not seeing the forest for the trees: novice programmers and the SOLO taxonomy", *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education,* Bologna, Italy, pp. 118-122, 2006.

[Logan 1988] Logan, G.D., "Toward an instance theory of automatization", *Psychol. Rev.,* vol. 95, 4, pp. 492-527, 1988.

[Luker 1994] Luker, P.A., "There's more to OOP than syntax!", *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education,* Phoenix, Arizona, United States, pp. 56-60, 1994.

[Madsen et al. 1993] Madsen, O.L., Møller-Pedersen, B. and Nygaard, K., *Object-Oriented Programming in the BETA Programming Language,* Addison-Wesley, pp. 337. 1993.

[Madsen et al. 1994] Madsen, K.H. and Trigg, R., "Introducing object-oriented technology in the humanities". In *Object-Oriented Software Development Environments: The Mjølner Approach,* Hertfordshire, Great Britain, Prentice-Hall, pp. 597-599, 1994.

[Malan et al. 2004] Malan, K. and Halland, K., "Examples that can do harm in learning programming", *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications,* Vancouver, BC, CANADA, pp. 83-87, 2004.

[Malmi et al. 2005] Malmi, L., Karavirta, V., Korhonen, A. and Nikander, J., "Experiences on automatically assessed algorithm simulation exercises with different resubmission policies", *J. Educ. Resour. Comput.,* vol. 5, 3, pp. 7, 2005.

[Malmi et al. 2007] Malmi, L. and Korhonen, A., "Activating Learning and Examination Methods in a Data Structures and Algorithms Course". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

[Margolis et al. 2002] Margolis, J. and Fisher, A., *Unlocking the Clubhouse: Women in Computing,* Cambridge Massachusetts, MIT Press, pp. 172. 2002.

[Martin 2003] Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices,* Upper Saddle River, NJ, Prentice-Hall, pp. 529. 2003.

[Marton et al. 1997] Marton, F. and Booth, S., *Learning and Awareness,* Mahwah, NJ, Lawrence Erlbaum Associates, pp. 2224. 1997.

[Mayer 1981] Mayer, R.E., "The Psychology of How Novices Learn Computer Programming", *ACM Comput. Surv.,* vol. 13, 1, pp. 121-141, 1981.

[Mayer et al. 1986] Mayer, R.E., Dyck, J.L. and Vilberg, W., "Learning to program and learning to think: what's the connection?", *Commun ACM,* vol. 29, 7, pp. 605-610, 1986.

[Mayer et al. 2002] Mayer, R.E. and Moreno, R., "Aids to computer-based multimedia learning", *Learning and Instruction,* vol. 12, 1, pp. 107-119, 2002.

[Mayer et al. 2003] Mayer, R.E. and Moreno, R., "Nine Ways to Reduce Cognitive Load in Multimedia Learning", *Educational Psychologist,* vol. 38, 1; 1, pp. 43-52, 2003.

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

[McGettrick et al. 2005] McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G. and Mander, K., "Grand Challenges in Computing: Education--A Summary", *The Computer Journal,* vol. 48, 1, pp. 42-48, 2005.

[McKeithen et al. 1981] McKeithen, K.B., Reitman, J.S., Rueter, H.H. and Hirtle, S.C., "Knowledge organization and skill differences in computer programmers", *Cognitive Psychology,* vol. 13, 3, pp. 307-325, 1981.

[Mead et al. 2006] Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C.S. and Thomas, L., "A cognitive approach to identifying measurable milestones for programming skill acquisition", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 182-194, 2006.

[Means 1988] Means, H.W., "A content analysis of ten introduction to programming textbooks", *SIGCSE '88: Proceedings of the nineteenth SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 283-287, 1988.

[Meyer 1987] Meyer, B., "Eiffel: programming for reusability and extendibility", *SIGPLAN Not.,* vol. 22, 2, pp. 85-94, 1987.

[Meyer 1989] Meyer, B., "From Structured Programming to Object-Oriented Design: The Road to Eiffel", *Structured Programming,* vol. 10, 1, pp. 19-39, 1989.

[Meyer 1992] Meyer, B., "Applying 'design by contract'", *Computer,* vol. 25, 10, pp. 40-51, 1992.

[Meyer 1993] Meyer, B. "Toward an object-oriented curriculum", 1993.

[Meyer 1997] Meyer, B., *Object-oriented software construction. 2nd ed,* , vol. 2. udgave, Upper Saddle River, New Jersey, Prentice Hall, 1997.

[Miller 1956] Miller, G.A., "The magical number seven, plus or minus two: some limits on our capacity for processing information", *Psychol. Rev.,* vol. 63, 2, pp. 81-97, 1956.

[Mills 1971] Mills, H.D., "Top-Down Programming in Large Systems". In *Debugging Techniques in Large Systems,* Englewood Cliffs, NJ, Prentice-Hall, pp. 41-55, 1971.

[Mitchell et al. 2002] Mitchell, R. and McKim, J., *Design by Contract by Example,* Adison-Wesley, pp. 238. 2002.

[Moreno et al. 1999] Moreno, R. and Mayer, R.E., "Cognitive principles of multimedia learning: The role of modality and contiguity", *J. Educ. Psychol.,* vol. 91, 2, pp. 358-368, 1999.

[Moreno 2004] Moreno, R., "Decreasing Cognitive Load for Novice Students: Effects of Explanatory versus Corrective Feedback in Discovery-Based Multimedia", *Instructional Science,* vol. 32, 1-2, pp. 99-113, 2004.

[Morgan et al 1992] Morgan, C. and Vickers, T., (Eds.) *On the Refinement Calculus.* London, Springer-Verlag, 1992. pp. 159.

[Morgan 1994] Morgan, C., *Programming from Specifications,* Prentice-Hall, pp. 332. 1994.

[Morris 1987] Morris, J.M., "A theoretical basis for stepwise refinement and the programming calculus", *Science of Computer Programming,* vol. 9, 3, pp. 287-306, 1987.

[Mosley 2002] Mosley, P.H., "The Cognitive Complexities Confronting Developers Using Object Technology", PhDPace University2002.

[Mousavi et al. 1995] Mousavi, S.Y., Low, R. and Sweller, J., "Reducing cognitive load by mixing auditory and visual presentation modes", *J. Educ. Psychol.,* vol. 87, 2, pp. 319-334, 1995.

[Muller et al. 2004] Muller, O., Haberman, B. and Averbuch, H., "(An almost) pedagogical pattern for pattern-based problem-solving instruction", *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* Leeds, United Kingdom, pp. 102-106, 2004.

[Muller et al. 2005a] Muller, O. and Haberman, B., "Guidelines for a multiple-goal CS introductory course: algorithmic problem-solving woven into OOP", *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* Caparica, Portugal, pp. 356-356, 2005.

[Muller 2005b] Muller, O., "Pattern oriented instruction and the enhancement of analogical reasoning", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 57-67, 2005.

[Naps et al. 1997] Naps, T., Bergin, J., Jiménez-Peris, R., McNally, M.F., Patiño-Martínez, M., Proulx, V.K. and Tarhio, J., "Using the WWW as the delivery mechanism for interactive, visualization-based instructional modules (report of the ITiCSE '97 working group on visualization)", *ITiCSE-WGR '97: The supplemental proceedings of the conference on Integrating technology into computer science education: working group reports and supplemental proceedings,* Uppsala, Sweden, pp. 13-26, 1997.

[Naps et al. 2002] Naps, T.L., Rössling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velázquez-Iturbide, J.Á, "Exploring the role of visualization and engagement in computer science education", *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 131-152, 2002.

[Naps et al. 2003] Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rössling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R.J., Anderson, J., Fleischer, R., Kuittinen, M. and McNally, M., "Evaluating the educational impact of visualization", *ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education,* Thessaloniki, Greece, pp. 124-136, 2003.

[Naps 2006] Naps, T. "JHAVÉ", http://jhave.org/, last updated: 2006, accessed: 2007.

[Naur 1972] Naur, P., "An experiment on program development", *BIT Numerical Mathematics,* vol. 12, 3, pp. 347-365, 1972.

[Naur 1992a] Naur, P., "Prospects for the Programming Methodologies (1981)". In *Computing: A Human Activity,* ACM Press, pp. 387-393, 1992.

[Naur 1992b] Naur, P., "Project Activity in Computer Science Education (1970)". In *Computing: A Human Activity,* ACM Press, pp. 228-239, 1992.

[Naur 1966] Naur, P., "Proof of algorithms by general snapshots", *BIT Numerical Mathematics,* vol. 6, 4, pp. 310-316, 1966.

[Netbeans 2006] Netbeans. "The NetBeans IDE 5.0 BlueJ Edition", http://edu.netbeans.org/bluej/, last updated: 2006, accessed: 2007.

[Newell et al. 1972] Newell, A. and Simon, H., *Human Problem Solving,* Englewood Cliffs, NJ, Prentice-Hall, 1972.

[Newell et al. 1981] Newell, A. and Rosenbloom, P., "Mechanisms of Skill Acquisition and the Law of Practice". In *Cognitive Skills and Their Acquisition,* Hillsdale, NJ, Erlbaum, pp. 1-56, 1981.

[Newell et al. 1989] Newell, A., Rosenbloom, P.S. and Laird, J.E., "Symbolic Architectures for Cognition". In *Foundations of Cognitive Science,* MIT Press, pp. 93-131, 1989.

[Newell 1990] Newell, A., *Unified Theories of Cognition,* Cambridge, MA, Harvard University Press, 1990.

[Newsted 1975] Newsted, P.R., "Grade and ability predictions in an introductory programming course", *SIGCSE Bull,* vol. 7, 2, pp. 87-91, 1975.

[Nguyen et al. 1999] Nguyen, D. and Wong, S.B., "Patterns for decoupling data structures and algorithms", *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education,* New Orleans, Louisiana, United States, pp. 87-91, 1999.

[Nguyen et al. 2001] Nguyen, D.Z. and Wong, S.B., "OOP in introductory CS: Better students through abstraction", *Procedings of the Fifth Workshop on and Tools for Assimilating Object-Oriented Concepts, OOPSLA '01,* Tampa, Florida, 2001.

[Norman 1981] Norman, D.A., "Categorization of action slips", *Psychol. Rev.,* vol. 88, 1, pp. 1-15, 1981.

[Norman et al. 1996] Norman, D.A. and Spohrer, J.C., "Learner-centered education", *Commun ACM,* vol. 39, 4, pp. 24-27, 1996.

[Nourie 2002] Nourie, D., "Teaching java technology with BlueJ", *Sun Developer Network,* http://java.sun.com/features/2002/07/bluej.html, 2002.

[Nowaczyk 1983] Nowaczyk, R.H., "Cognitive Skills Needed in Computer Programming", Atlanta, GA, USA, pp. 1-14, 1983.

[Object Mentor Inc. 2006] Object Mentor Inc. "JUnit.org", www.junit.org, last updated: 2006, accessed: 2007.

[Or-Bach et al. 2004] Or-Bach, R. and Lavy, I., "Cognitive activities of abstraction in object orientation: an empirical study", *SIGCSE Bull,* vol. 36, 2, pp. 82-86, 2004.

[Paas 1992] Paas, F.G., "Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach", *J. Educ. Psychol.,* vol. 84, 4, pp. 429-434, 1992.

[Paas et al. 1994] Paas, F.G.W.C. and Van Merriënboer, J.J.G., "Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach", *J. Educ. Psychol.,* vol. 86, 1, pp. 122-133, 1994.

[Paas et al. 2003] Paas, F., Renkl, A. and Sweller, J., "Cognitive Load Theory and Instructional Design: Recent Developments", *Educational Psychologist,* vol. 38, 1; 1, pp. 1-4, 2003.

[Paas et al. 2004] Paas, F., Renkl, A. and Sweller, J., "Cognitive Load Theory: Instructional Implications of the Interaction between Information Structures and Cognitive Architecture", *Instructional Science,* vol. 32, 1-2, pp. 1-8, 2004.

[Papert 1993] Papert, S., *Mindstorms: children, computers, and powerful ideas,* New York, Basic Books, pp. 230. 1993.

[Papert 2007] Papert, S. "Seymour Papert", http://en.wikipedia.org/wiki/Seymour_Papert, last updated: 2007, accessed: 2007.

[Patterson et al. 2003] Patterson, A., Kölling, M. and Rosenberg, J., "Introducing unit testing with BlueJ", *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education,* Thessaloniki, Greece, pp. 11-15, 2003.

[Pattis 1990] Pattis, R.E., "A philosophy and example of CS-1 programming projects", *SIGCSE '90: Proceedings of the twenty-first SIGCSE technical symposium on Computer science education,* Washington, D.C., United States, pp. 34-39, 1990.

[Pattis 1993] Pattis, R.E., "The “procedures early” approach in CS 1: a heresy", *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education,* Indianapolis, Indiana, United States, pp. 122-126, 1993.

[Pecinovský et al. 2006] Pecinovský, R., Pavlícková, J. and Pavlícek, L., "Let's modify the objects-first approach into design-patterns-first", *ITICSE '06: Proceedings of the 11th an-*

nual SIGCSE conference on Innovation and technology in computer science education, Bologna, Italy, pp. 188-192, 2006.

[Pedroni 2003] Pedroni, M., "Teaching Introductory Programming with the Inverted Curriculum Approach", DiplomaETH Zürichpp. 1-86, 2003.

[Pedroni et al. 2006] Pedroni, M. and Meyer, B., "The inverted curriculum in practice", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 481-485, 2006.

[Penney 1989] Penney, C.G., "Modality effects and the structure of short term verbal memory", *Memory & Cognition,* vol. 17, pp. 398-422, 1989.

[Perkins 1981] Perkins, D.N., *The Mind's Best Work,* Harvard University Press, pp. 324. 1981.

[Petre et al. 2005] Petre, M. and Green, T., "Editorial", *Computer Science Education,* vol. 15, 1, pp. 3-5, 2005.

[Phillips 1995] Phillips, D.C., "The Good, the Bad, and the Ugly: The Many Faces of Constructivism", *Educational Researcher,* vol. 24, 7, pp. 5-12, 1995.

[Piaget 2007] Piaget, J. "Jean Piaget", http://en.wikipedia.org/wiki/Jean_Piaget, last updated: 2007, accessed: 2007.

[Pirolli et al. 1985] Pirolli, P. and Anderson, J.R., "The role of learning from examples in the acquisition of recursive programming skills", *Canadian Journal of Psychology,* vol. 39, pp. 240-272, 1985.

[Pirolli 1991] Pirolli, P., "Effects of Examples and Their Explanations in a Lesson n Recursion: A Production System Analysis", *Cognition & Instruction,* vol. 8, 3, pp. 207, 1991.

[Pirolli et al. 1994] Pirolli, P. and Recker, M., "Learning Strategies and Transfer in the Domain of Programming", *Cognition and Instruction,* vol. 12, 3, pp. 235-275, 1994.

[Pollock et al. 2002] Pollock, E., Chandler, P. and Sweller, J., "Assimilating complex information", *Learning and Instruction,* vol. 12, 1, pp. 61-86, 2002.

[Polya 1957] Polya, G., *How to Solve It,* New Jersey, Princeton University Press, pp. 253. 1957.

[Posner 1993] Posner, M.I., (Eds.) *Foundations in Cognitive Science.* MIT Press, 1993. pp. 888.

[PPIG 2005] PPIG. "Psychology of Programming Interest Group", http://www.ppig.org/, last updated: 2005, accessed: 2006.

[Preiss 1999] Preiss, B.R., "Design patterns for the data structures and algorithms course", *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education,* New Orleans, Louisiana, United States, pp. 95-99, 1999.

[Proulx et al. 2002] Proulx, V.K., Raab, J. and Rasala, R., "Objects from the beginning - with GUIs", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 65-69, 2002.

[Proulx et al. 2006] Proulx, V.K. and Gray, K.E., "Design of class hierarchies: an introduction to OO program design", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 288-292, 2006.

[PVW 2000] PVW. "Program Visualization Workshop", http://www.cs.joensuu.fi/pages/pvw/workshop.htm, last updated: 2000, accessed: 2007.

[PVW 2002] PVW. "Second Program Visualization Workshop", http://stwww.weizmann.ac.il/G-CS/BENARI/pvw/pvw.html, last updated: 2002, accessed: 2007.

[PVW 2004] PVW. "Third Program Visualization Workshop", http://www.dcs.warwick.ac.uk/pvw04/, last updated: 2004, accessed: 2007.

[PVW 2006] PVW. "Fourth Program Visualization Workshop", http://www.algoanim.net/pvw2006/, last updated: 2006, accessed: 2007.

[Quilici et al. 1996] Quilici, J.L. and Mayer, R.E., "Role of examples in how students learn to categorize statistics word problems", *J. Educ. Psychol.,* vol. 88, 1, pp. 144-161, 1996.

[Raadt et al. 2004a] Raadt, M.d., Watson, R. and Toleman, M., "Introductory programming: what's happening today and will there be any students to teach tomorrow?", *ACE '04: Proceedings of the sixth conference on Australasian computing education,* Dunedin, New Zealand, pp. 277-282, 2004.

[Raadt et al. 2004b] Raadt, M.d., Toleman, M. and Watson, R., "Training strategic problem solvers", *SIGCSE Bull,* vol. 36, 2, pp. 48-51, 2004.

[Ragonis et al. 2005a] Ragonis, N. and Ben-Ari, M., "On understanding the statics and dynamics of object-oriented programs", *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, pp. 226-230, 2005.

[Ragonis et al. 2005b] Ragonis, N. and Ben-Ari, M., "A long-term investigation of the comprehension of OOP concepts by novices", *Computer Science Education,* vol. 15, 3, pp. 203-221, 2005.

[Ramalingam et al. 2004] Ramalingam, V., LaBelle, D. and Wiedenbeck, S., "Self-efficacy and mental models in learning to program", *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,* Leeds, United Kingdom, pp. 171-175, 2004.

[Rasala et al. 2001] Rasala, R., Raab, J. and Proulx, V.K., "Java power tools: model software for teaching object-oriented design", *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education,* Charlotte, North Carolina, United States, pp. 297-301, 2001.

[Rauchas et al. 2006] Rauchas, S., Rosman, B., Konidaris, G. and Sanders, I., "Language performance at high school and success in first year computer science", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 398-402, 2006.

[Reed 1998] Reed, D., "Incorporating problem-solving patterns in CS1", *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 6-9, 1998.

[Reek 1995] Reek, M.M., "A top-down approach to teaching programming", *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education,* Nashville, Tennessee, United States, pp. 6-9, 1995.

[Renkl et al. 2003] Renkl, A. and Atkinson, R.K., "Structuring the Transition From Example Study to Problem Solving in Cognitive Skill Acquisition: A Cognitive Load Perspective", *Educational Psychologist,* vol. 38, 1; 1, pp. 15-22, 2003.

[Rich et al. 2004] Rich, L., Perry, H. and Guzdial, M., "A CS1 course designed to address interests of women", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 190-194, 2004.

[Riel 1996] Riel, A.J., *Object-Oriented Design Heuristics,* Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc, 1996.

[Riley 1981] Riley, D.D., "Teaching problem solving in an introductory computer science class", *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, United States, pp. 244-251, 1981.

[Rist 1989] Rist, R.S., "Schema creation in programming", *Cognitive Science,* vol. 13, 3, pp. 389-414, 1989.

[Roberts 2004a] Roberts, E., "The dream of a common language: the search for simplicity and stability in computer science education", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 115-119, 2004.

[Roberts 2004b] Roberts, E., "Resources to support the use of Java in introductory computer science", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 233-234, 2004.

[Roberts 2006] Roberts, T.S., "The use of multiple choice tests for formative and summative assessment", *ACE '06: Proceedings of the 8th Austalian conference on Computing education,* Hobart, Australia, pp. 175-180, 2006.

[Roberts et al. 2006] Roberts, E., Bruce, K., Cutler, R., Cross, J., Grissom, S., Klee, K., Rodger, S., Trees, F., Utting, I. and Yellin, F. "The ACM Java Task Force Version 1.0", http://jtf.acm.org/, last updated: 2006, accessed: 2007.

[Robillard 2005] Robillard, P.N., "Opportunistic Problem Solving in Software Engineering", *IEEE Softw.,* vol. 22, 6, pp. 60-67, 2005.

[Robins et al. 2003] Robins, A., Rountree, J. and Rountree, N., "Learning and Teaching Programming: A Review and Discussion", *Journal of Computer Science Education,* vol. 13, 2, pp. 137-172, 2003.

[Rodger et al. 2006a] Rodger, S.H., Bressler, B., Finley, T. and Reading, S., "Turning automata theory into a hands-on course", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 379-383, 2006.

[Rodger 2006b] Rodger, S., "Learning automata and formal languages interactively with JFLAP", *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education,* Bologna, Italy, pp. 360-360, 2006.

[Rosenberg et al. 1997] Rosenberg, J. and Kölling, M., "Testing object-oriented programs: making it simple", *SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education,* San Jose, California, United States, pp. 77-81, 1997.

[Rössling et al. 2006] Rössling, G., Naps, T., Hall, M.S., Karavirta, V., Kerren, A., Leska, C., Moreno, A., Oechsle, R., Rodger, S.H., Urquiza-Fuentes, J. and Velázquez-Iturbide, J.Á, "Merging interactive visualizations with hypertextbooks and course management", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 166-181, 2006.

[Rosson et al. 1996] Rosson, M.B. and Carroll, J.M., "Scaffolded examples for learning object-oriented design", *Commun ACM,* vol. 39, 4, pp. 46-47, 1996.

[Roumani 2006] Roumani, H., "Practice what you preach: full separation of concerns in CS1/CS2", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education,* Houston, Texas, USA, pp. 491-494, 2006.

[Rountree et al. 2002] Rountree, N., Rountree, J. and Robins, A., "Predictors of success and failure in a CS1 course", *SIGCSE Bull,* vol. 34, 4, pp. 121-124, 2002.

[Rubinstein 1975] Rubinstein, M., *Patterns of Problem Solving,* Englewood Cliffs, NJ, Prentice-Hall, pp. 544. 1975.

[Saiedian 2001] Saiedian, H., "Practical Software Engineering Education", *Computer Science Education,* vol. 11, 1, pp. 3-5, 2001.

[Sajaniemi et al. 2003] Sajaniemi, J. and Kuittinen, M., "Program animation based on the roles of variables", *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization,* San Diego, California, pp. 7-ff, 2003.

[Sajaniemi 2006] Sajaniemi, J. "The Roles of Variables Home Page", http://cs.joensuu.fi/~saja/var_roles/, last updated: 2006, accessed: 2007.

[Schmidt 1980] Schmidt, E.M. "Lecture notes for CS1 (in Danish)", 1980.

[Schmolitzky 2005] Schmolitzky, A., "Towards Complexity Levels of Object Systems Used in Software Engineering Education", Glasgow, UK, 2005.

[Schnotz et al. 2005] Schnotz, W. and Rasch, T., "Enabling, Facilitating, and Inhibiting Effects of Animations in Multimedia Learning: Why Reduction of Cognitive Load Can Have Negative Results on Learning", *Educational Technology Research & Development,* vol. 53, 3; 3, pp. 47-58, 2005.

[Schoenfeld 1981] Schoenfeld, A.H., "Episodes and executive decisions in mathematical problem solving", 1981.

[Scholtz et al. 1992] Scholtz, J. and Wiedenbeck, S., "The role of planning in learning a new programming language", *International Journal of Man-Machine Studies,* vol. 37, 2, pp. 191-214, 1992.

[Shackelford et al. 1993] Shackelford, R.L. and Badre, A.N., "Why can't smart students solve simple programming problems?", *International Journal of Man-Machine Studies,* vol. 38, 6, pp. 985-997, 1993.

[Shayer et al. 1981] Shayer, M. and Adey, P., *Towards a science of science teaching. cognitive development and curriculum demand,* Oxford, Heinemann Educational, 1981.

[Sheil 1981] Sheil, B.A., "The Psychological Study of Programming", *ACM Comput. Surv.,* vol. 13, 1, pp. 101-120, 1981.

[Shneiderman 1976] Shneiderman, B., "Exploratory experiments in programmer behavior", *International Journal of Parallel Programming,* vol. 5, 2, pp. 123-143, 1976.

[Sicilia 2006] Sicilia, M., "Strategies for teaching object-oriented concepts with Java", *Journal of Computer Science Education,* vol. 16, 1, pp. 1-18, 2006.

[SIGCSE 2006] SIGCSE. "Special Interest Group in Computer Science Education", http://www.sigcse.org/, last updated: 2006, accessed: 2006.

[Simon 1973] Simon, H.A., "The Organization of Complex Systems". In *Hierarchy Theory: The Challenge of Complex Systems,* New York, George Braziller, pp. 1-27, 1973.

[Simons et al. 1991] Simons, B., Frailey, D.J., Turner, A.J., Zweben, S.H. and Denning, P.J., "An ACM response: the scope and directions of Computer Science", *Commun ACM,* vol. 34, 10, pp. 121-131, 1991.

[Singley et al. 1989] Singley, M. and Anderson, J.R., *The Transfer of Cognitive Skill,* Harvard University Press, 1989.

[Skublics et al. 1991] Skublics, S. and White, P., "Teaching Smalltalk as a first programming language", *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education,* San Antonio, Texas, United States, pp. 231-234, 1991.

[Smith et al. 1993] Smith, J.P.,III, diSessa, A.A. and Roschelle, J., "Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition", *The Journal of the Learning Sciences,* vol. 3, 2, pp. 115-163, 1993.

[SoftVis 2006] SoftVis. "ACM Symposium on Software Visualization", http://www.softvis.org, last updated: 2006, accessed: 2007.

[Soloway et al. 1983] Soloway, E., Bonar, J. and Ehrlich, K., "Cognitive strategies and looping constructs: an empirical study", *Commun ACM,* vol. 26, 11, pp. 853-860, 1983.

[Soloway 1986] Soloway, E., "Learning to program = learning to construct mechanisms and explanations", *Commun ACM,* vol. 29, 9, pp. 850-858, 1986.

[Spohrer et al. 1986] Spohrer, J.C. and Soloway, E., "Novice mistakes: are the folk wisdoms correct?", *Commun ACM,* vol. 29, 7, pp. 624-632, 1986.

[Soloway et al. 1989] Soloway, E. and Spohrer, J.C., *Studying the novice programmer,* Hillsdale, N.J., Lawrence Erlbaum, 1989.

[Sprague et al. 2002] Sprague, P. and Schahczenski, C., "Abstraction the key to CS1", *J. Comput. Small Coll.,* vol. 17, 3, pp. 211-218, 2002.

[Stamouli et al. 2006] Stamouli, I. and Huggard, M., "Object oriented programming and program correctness: the students' perspective", *ICER '06: Proceedings of the 2006 international workshop on Computing education research,* Canterbury, United Kingdom, pp. 109-118, 2006.

[Standage 1998] Standage, T., *The Victorian Internet,* New York, Berkeley Publishing Group, pp. 227. 1998.

[Stasko et al. 2004] Stasko, J.T. and Hundhausen, C.D., "Algorithm Visualization". In *Computer Science Education Research,* Taylor & Francis, 2004.

[Stein 1998] Stein, L.A., "What We Swept Under the Rug: Radically Rethinking CS1", *Computer Science Education,* vol. 8, 2, pp. 118-129, 1998.

[Stein 2003] Stein, L.A. "Interactive Programming in Java", http://www.cs101.org/ipij/, last updated: 2003, accessed: 2007.

[Stein 2002] Stein, M.V., "Mathematical preparation as a basis for success in CS-II", *J. Comput. Small Coll.,* vol. 17, 4, pp. 28-38, 2002.

[Stiggins 2005] Stiggins, R.J., *Student-Involved Assessment for Learning,* Upper Saddle River, NJ, Prentice-Hall, pp. 400. 2005.

[Stroustrup 1985] Stroustrup, B., "A C++ tutorial", *ACM '85: Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective,* Denver, Colorado, United States, pp. 56-64, 1985.

[Stroustrup 2000] Stroustrup, B., *The C++ Programming Language,* Addison-Wesley, pp. 1040. 2000.

[Sweller et al. 1982] Sweller, J., Mawer, R.F. and Howe, W., "Consequences of History-Cued and Means-End Strategies in Problem Solving", *Am. J. Psychol.,* vol. 95, 3, pp. 455-483, 1982.

[Sweller et al. 1983] Sweller, J., Mawer, R.F. and Ward, M.R., "Development of expertise in mathematical problem solving", *J. Exp. Psychol. : Gen.,* vol. 112, 4, pp. 639-661, 1983.

[Sweller et al. 1985] Sweller, J. and Cooper, G.A., "The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra", *Cognition and Instruction,* vol. 2, 1, pp. 59-89, 1985.

[Sweller 1988] Sweller, J., "Cognitive load during problem solving: Effects on learning", *Cognitive Science,* vol. 12, 2, pp. 257-285, 1988.

[Sweller et al. 1990] Sweller, J., Chandler, P., Tierney, P. and Cooper, M., "Cognitive load as a factor in the structuring of technical material", *J. Exp. Psychol. : Gen.,* vol. 119, 2, pp. 176-192, 1990.

[Sweller 1994a] Sweller, J., "Cognitive load theory, learning difficulty, and instructional design", *Learning and Instruction,* vol. 4, 4, pp. 295-312, 1994.

[Sweller et al. 1994b] Sweller, J. and Chandler, P., "Why Some Material Is Difficult to Learn", *Cognition and Instruction,* vol. 12, 3, pp. 185-233, 1994.

[Tarmizi et al. 1988] Tarmizi, R.A. and Sweller, J., "Guidance during mathematical problem solving", *J. Educ. Psychol.,* vol. 80, 4, pp. 424-436, 1988.

[Taylor 1977] Taylor, R.P., "Teaching programming to beginners", *SIGCSE '77: Proceedings of the seventh SIGCSE technical symposium on Computer science education,* Atlanta, Georgia, United States, pp. 88-92, 1977.

[Teif et al. 2006] Teif, M. and Hazzan, O., "Partonomy and taxonomy in object-oriented thinking: junior high school students' perceptions of object-oriented basic concepts", *ITiCSE-WGR '06: Working group reports on ITiCSE on Innovation and technology in computer science education,* Bologna, Italy, pp. 55-60, 2006.

[Temte 1991] Temte, M.C., "Let's begin introducing the object-oriented paradigm", *SIGCSE '91: Proceedings of the twenty-second SIGCSE technical symposium on Computer science education,* San Antonio, Texas, United States, pp. 73-77, 1991.

[Thompson 2006] Thompson, E. "Researching learning to program", http://www.massey.ac.nz/~elthomps/, last updated: 2006, accessed: 2007.

[Trafton et al. 1993] Trafton, J.G. and Reiser, B.J., "The contributions of studying examples and solving problems to skill acquisition", *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society,* pp. 1017-1022, 1993.

[Traynor et al. 2006] Traynor, D., Bergin, S. and Gibson, J.P., "Automated assessment in CS1", *ACE '06: Proceedings of the 8th Austalian conference on Computing education,* Hobart, Australia, pp. 223-228, 2006.

[Tu et al. 1990] Tu, J. and Johnson, J.R., "Can computer programming improve problem-solving ability?", *SIGCSE Bull,* vol. 22, 2, pp. 30-33, 1990.

[Tucker 1996] Tucker, A.B., "Strategic directions in computer science education", *ACM Comput. Surv.,* vol. 28, 4, pp. 836-845, 1996.

[Turner 1991] Turner, A.J., "Computing Curricula 1991", *Commun ACM,* vol. 34, 6, pp. 68-84, 1991.

[Tyler 1949] Tyler, R.W., *Basic Principles of Curriculum and Instruction,* Chicago, The University of Chicago Press, pp. 134. 1949.

[Ulloa 1980] Ulloa, M., "Teaching and learning computer programming: a survey of student problems, teaching methods, and automated instructional tools", *SIGCSE Bull,* vol. 12, 2, pp. 48-64, 1980.

[UML 2007] UML. "Unified Modeling Language", http://en.wikipedia.org/wiki/Unified_Modeling_Language, last updated: 2007, accessed: 2007.

[van Merriënboer et al. 2003] van Merriënboer, J.J.G., Kirschner, P.A. and Kester, L., "Taking the Load Off a Learner's Mind: Instructional Design for Complex Learning", *Educational Psychologist,* vol. 38, 1; 1, pp. 5-13, 2003.

[van Merriënboer et al. 2005] van Merriënboer, J.J.G. and Ayres, P., "Research on Cognitive Load Theory and Its Design Implications for E-Learning", *Educational Technology Research & Development,* vol. 53, 3, pp. 5-13, 2005.

[VanLehn 1989] VanLehn, K., "Problem Solving and Cognitive Skill Acquisition". In *Foundations of Cognitive Science,* MIT Press, pp. 527-579, 1989.

[VanLehn 1996] VanLehn, K., "Cognitive Skill Acquisition", *Annual Review of Psychology,* vol. 47, pp. 513-539, 1996.

[Ventura 2003] Ventura, P.R., "On the origins of programmers: Identifying predictors of success for an objects first CS1", Ph.D. thesisThe State University of New York at Buffalo, Buffalo, New York U.S.A.2003.

[Ventura et al. 2004] Ventura, P.R. and Ramamurthy, B., "Wanted: CS1 students. no experience required", *Proceedings of the 35th SIGCSE technical symposium on Computer science education,* Norfolk, Virginia, USA, pp. 240-244, 2004.

[Ventura 2005] Ventura, P.R., "Identifying predictors of success for an objects-first CS1.", *Computer Science Education,* vol. 15, 3, pp. 223-243, 2005.

[Wallingford 1996] Wallingford, E., "Toward a first course based on object-oriented patterns", *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education,* Philadelphia, Pennsylvania, United States, pp. 27-31, 1996.

[Wallingford 2000] Wallingford, E., "Using patterns in the CS curriculum", *CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges,* Ramapo College of New Jersey, Mahwah, New Jersey, United States, pp. 235-237, 2000.

[Ward et al. 1990] Ward, M. and Sweller, J., "Structuring Effective Worked Examples", *Cognition and Instruction,* vol. 7, 1, pp. 1-39, 1990.

[Wegner et al. 1996] Wegner, P. and Doyle, J., "Editorial: Strategic directions in computing research", *ACM Comput. Surv.,* vol. 28, 4, pp. 565-574, 1996.

[Weiner 1978] Weiner, L.H., "The roots of structured programming", *Papers of the SIGCSE/CSA technical symposium on Computer science education,* Detroit, Michigan, pp. 243-254, 1978.

[Werth 1986] Werth, L.H., "Predicting student performance in a beginning computer science class", *SIGCSE Bull,* vol. 18, 1, pp. 138-143, 1986.

[Westfall 2001] Westfall, R., "Technical opinion: Hello, world considered harmful", *Commun ACM,* vol. 44, 10, pp. 129-130, 2001.

[Wiedenbeck et al. 1993] Wiedenbeck, S., Fix, V. and Scholtz, J., "Characteristics of the mental representations of novice and expert programmers: an empirical study", *Int. J. Man-Mach. Stud.,* vol. 39, 5, pp. 793-812, 1993.

[Wiedenbeck 2005] Wiedenbeck, S., "Factors affecting the success of non-majors in learning to program", *ICER '05: Proceedings of the 2005 international workshop on Computing education research,* Seattle, WA, USA, pp. 13-24, 2005.

[Wikipedia 2007a] Wikipedia. "Hawthorne effect", http://en.wikipedia.org/wiki/Hawthorne_effect, last updated: 2007, accessed: 2007.

[Wikipedia 2007b] Wikipedia. "Experience curve effects", http://en.wikipedia.org/wiki/Experience_curve_effects, last updated: 2007, accessed: 2007.

[Wikipedia 2007c] Wikipedia. "Advanced Placement Program", http://en.wikipedia.org/wiki/Advanced_Placement_Program, last updated: 2007, accessed: 2007.

[Wilkerson et al. 2005] Wilkerson, M., Griswold, W.G. and Simon, B., "Ubiquitous presenter: increasing student access and control in a digital lecturing environment", *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, pp. 116-120, 2005.

[Williams et al. 2001] Williams, L.A. and Kessler, R.R., "Experiments with Industry's “Pair-Programming” Model in the Computer Science Classroom", *Computer Science Education,* vol. 11, 1, pp. 7-20, 2001.

[Williams et al. 2002] Williams, L. and Tomayko, J., "Agile Software Development", *Computer Science Education,* vol. 12, 3, pp. 167, 2002.

[Willshire 1995] Willshire, M.J., "Old dogs, new tricks", *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education,* Nashville, Tennessee, United States, pp. 178-181, 1995.

[Wilson et al. 2001] Wilson, B.C. and Shrock, S., "Contributing to success in an introductory computer science course: a study of twelve factors", *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education,* Charlotte, North Carolina, United States, pp. 184-188, 2001.

[Wilson 2002] Wilson, B.C., "A Study of Factors Promoting Success in Computer Science Including Gender Differences", *Computer Science Education,* vol. 12, 1/2, pp. 141, 2002.

[Winslow 1996] Winslow, L.E., "Programming pedagogy — a psychological overview", *SIGCSE Bull,* vol. 28, 3, pp. 17-22, 1996.

[Wirth 1971] Wirth, N., "Program development by stepwise refinement", *Commun ACM,* vol. 14, 4, pp. 221-227, 1971.

[Wirth 1974] Wirth, N., "On the Composition of Well-Structured Programs", *ACM Comput. Surv.,* vol. 6, 4, pp. 247-259, 1974.

[Wirth 2002] Wirth, N., "Computing science education: the road not taken", *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education,* Aarhus, Denmark, pp. 1-3, 2002.

[Wittgenstein 1953] Wittgenstein, L., *Philosophical Investigations,* Basil Blackwell, 1953.

[Wolz et al. 1994] Wolz, U. and Conjura, E., "Integrating mathematics and programming into a three tiered model for computer science education", *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education,* Phoenix, Arizona, United States, pp. 223-227, 1994.

[Woodford et al. 2005] Woodford, K. and Bancroft, P., "Multiple choice questions not considered harmful", *ACE '05: Proceedings of the 7th Australasian conference on Computing education,* Newcastle, New South Wales, Australia, pp. 109-116, 2005.

[Zhu et al. 1987] Zhu, X. and Simon, H.A., "Learning Mathematics From Examples and by Doing", *Cognition & Instruction,* vol. 4, 3, pp. 137, 1987.

II Papers

12 Potential Success Factors

The paper *An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course* presented in this chapter has been published as a conference paper [Bennedsen et al. 2005b].

[Bennedsen et al. 2005b] Bennedsen, J. and Caspersen, M.E., “An investigation of potential success factors for an introductory model-driven programming course”, *ICER '05: Proceedings of the 2005 International workshop on Computing Education Research,* Seattle, WA, USA, pp. 155-163, 2005.

An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course

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

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

ABSTRACT

In order to improve the course design of a CS1 model-driven programming course we study potential indicators of success for such a course. We explain our specific interpretation of objects-first.

Of eight potential indicators of success, we have found only two to be significant at a 95% confidence interval: math grade from high school and course work. The two significant indicators explain 24.2% of the variation of the exam grade. The result concerning math grade contradicts earlier findings.

We discuss four aspects of our research: the explanation power of the potential success indicators, the impact of our findings on teaching, limits of what to conclude from the available data, and the variety of the notion “objects-first”.

Because of the variety of interpretations of “objects-first”, the present research is necessary as a supplement to earlier research in order to make generalizable results on the success factors for objects-first programming.

Categories and Subject Descriptors

K3.2 [**Computers & Education**]: Computer and Information Science Education – *computer science education, information systems education.*

General Terms: Experimentation, Human Factors.

Keywords: Objects-first, CS1, object-oriented programming, model-driven programming, predictors of success, course design.

1. INTRODUCTION

A substantial amount of research has been conducted in order to identify variables that are predictors of success of students aiming for a university degree. Investigated variables encompass gender [23], the educational level of parents [26], ACT/SAT scores [5, 12, 23], and emotional factors [25]. Research has been conducted in the general context of education, within computer science, and in the more topic specific area of introductory programming [4, 6, 13, 17, 19]. Even in the area of introductory object-oriented programming there has been research trying to establish general factors to predict success or failure of particular students. Especially the work of Phil Venture [27] focuses on a systematic evaluation of hypothesis related to the factors for success of an introductory programming course using an objects-first approach [14]. The results are documented in [27, 28].

Ventura analysed different factors and their influence on the outcome of participation in an object-first CS1 course. The predictors included prior programming experience, mathematical ability, academic and psychological variables, gender, and measures of student effort [27 p. xxi]. Ventura’s conclusion is that there are big differences between the previous findings in imperative-first programming courses and his object-first programming course. In the studies of imperative-first courses the student’s mathematical abilities was found to be a predictor of success [17]. Ventura [27], however, found that this is not the case in his object-first CS1 course. In the imperative-first course, prior programming experience was also a predictor for success; Ventura found this was not the case in his objects-first course. As a curiosity he found that previous knowledge of Java was a negative predictor of success: the students with previous knowledge performed worse than students without [27 p. 73].

As always there are some preconditions to the research. One important precondition is the characteristics of the course that founded the basis for the research. Ventura used a CS1 course with a graphics early approach. In [28] he describes the graphics early approach as follows: The course focuses primarily on the teaching of problem solving using object-oriented design techniques with the following features [28 p. 241]:

Design-centered. Through the introduction of a simplified version of UML class diagrams, students are taught to think about problem solutions independently of the code. Design once, code anywhere has become the motto for the class. Design patterns are introduced both in lecture and integrated into the programming assignments. These serve as examples of good design as well as vehicles to encourage students to think at a higher level of abstraction.

Graphical. Classroom examples use graphics to motivate and ground OO concepts such as encapsulation, inheritance, and polymorphism. The programming assignments are also graphical allowing the students to build programs that are like those they are used to using.

Objects-first. Students are taught from the very beginning to think in terms of objects and the fundamentals of object-oriented programming, encapsulation, inheritance, and polymorphism. These

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICER’05, October 1–2, 2005, Seattle, Washington, USA.
Copyright 2005 ACM 1-59593-043-4/05/0010...$5.00.

concepts are introduced before traditional language constructs for selection and iteration.

Furthermore, Ventura writes: “Empirical testing was conducted on the graphical design-centric objects-first CS1 to identify the predictors of success” [28 p. 127]. Venturas findings are only valid for students participating in an introductory programming course similar to his. In the current research, we look for potential success factors for an introductory programming using a different approach than Ventura’s; our approach is best characterized as a model-based approach to programming [1].

2. A MODEL-DRIVEN PROGRAMMING COURSE

This section describes goal, form, and content of the model-driven programming course as well as the lab test that constitutes the final examination and upon which the grading is based.

2.1 General Information

This course constitutes the first half of CS1 at University of Aarhus. The course runs for seven weeks, one to two weeks after the course there is a lab test with a binary pass/fail grading.

The grading is based solely upon the behaviour in and result of a lab test; suitable performance during the course is a prerequisite for the final exam but does not count as part of the grading.

There are approximately 235 students from a variety of study programmes, e.g. computer science, mathematics, geology, nano science, economy, multimedia, etc. 40% are majors in computer science, and they are the only group of students that continue with the second half of CS1. The rest of the students proceed to other programming courses related to their fields (e.g. multimedia programming, scientific computing, etc.).

The students are grouped in teams of 18-20 students; in the fall of 2004 there where 13 teams. Each team has its own Teaching Assistant (TA).

2.2 Goals

The purpose of the course is that the student learns the foundation for systematic construction of simple programs and through this obtains knowledge about the role of conceptual modelling in object-oriented programming.

Furthermore, it is the goal that the student becomes familiar with a modern programming language, fundamental programming language concepts, and selected class libraries.

After the course the student will be able to explain and use fundamental elements in a modern programming language, use conceptual modelling in relation to preparing simple object-oriented programs, implement simple OO-models in a modern programming language, and use selected class libraries.

2.3 Form

The course runs for seven weeks; every week there are four lecture hours[1], one lab hour with a TA, and three class hours also with a TA. Besides scheduled hours, the students are supposed to work approximately seven hours per week in study groups or on their own.

[1] For scheduled actvities (lectures, labs, classes, etc.) an hour means only 45 minutes.

Every week (except for the first), there is a mandatory assignment that must be handed in to the TA. The TA examines the assignments and gives personal as well as collective feedback to the students. If an assignment is too weak, the student gets a chance to improve it. Approval of five out of six weekly assignments is a prerequisite for the final exam but does not count as part of the grading.

The four lecture hours per week are used for presentation and discussion of general concepts and specific details in the course material, but also for live programming. Live programming is programming in front of the students in the lecture theatre using computer and projector. The purpose of live programming is to reveal the programming process to the students (see [2]).

The one lab hour per week is unstructured in the sense that the students (typically in pairs) work on what they find useful, the purpose of the lab is that the students can get help from a TA while working on the exercises of the week.

The three class hours per week are used for discussion of the weekly assignment, for discussion of other exercises that the students has been working on, as well as for discussion of topics from the textbook.

For the coming versions of the course, we are planning to adopt closed labs as a more structured form of the lab activities; also we are planning to reschedule such that two (or three) hours per week are spent on closed labs and two (or one) in the classroom.

2.4 Contents

The course content is fundamental programming language concepts, object-orientation, and techniques for systematic construction of simple programs.

- *Fundamental programming language concepts*. Variable, value, type, expression, object, class, encapsulation, control structure, method/procedure, recursion, type hierarchies.
- *Object-orientation*. Modelling; class structures (specialization, aggregation and association); use of selected class libraries (in particular collection libraries), interfaces and abstract classes.
- *Systematic development of small programs*. Modularization, stepwise refinement/incremental development, test.

The above is a logical listing of the course contents; it is *not* the order in which the content is covered. The content is covered using a spiral approach [3]; for further details on the structure and contents of the course, see [1, 7, 8].

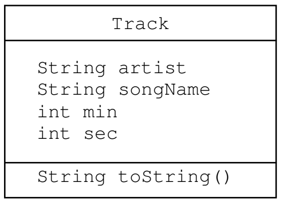
2.5 The Exam

The exam is organized such that 20 students are tested concurrently. The test takes place in a lab; besides the 20 students, five TAs, the lecturer and an external examiner are present in the lab.

We schedule one hour per group of 20 students, but only 30 minutes are used for the lab test. The rest of the time is used for administrative activities and as a buffer.

Lab Test Exercise (30 minute test)

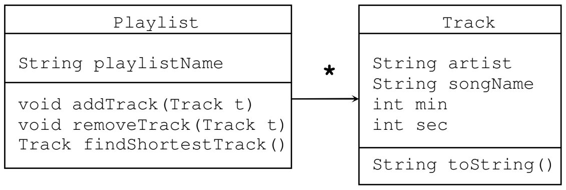
1. Create a class, *Track*, that represents a piece of music; the *Track* class is specified in the following UML diagram.



The four field variables must be initialized in a constructor (through four parameters of suitable types). The method *toString* must return a string representation for a piece of music, e.g.

```
"Yesterday: The Beatles (2:05)"
```

2. Create a test method named *exam* in class *Driver*. The method must be static, have return type void, and have no parameters.
3. Create two *Track* objects in the exam method using object references *t1* and *t2*; print the two *Track* objects using the *toString* method.
4. Create a new class, *Playlist*, representing a collection of *Track*s; the *Playlist* class and its relation to the *Track* class is specified in the following UML diagram:



5. Implement the method *addTrack* (and *removeTrack*) so that it adds (removes) the object *t* to (from) the *Playlist* object.
6. Create a *Playlist* object in the *exam* method in the *Driver* class; associate the two existing *Track* objects with the *Playlist* object.
7. Implement the method *findShortestTrack*. The method must return a shortest (measured in playing time) *Track* object from a *Playlist* object. You can assume a non-empty *Playlist* object. In other words: You need not worry about the *playlist* being empty.
8. Use the methods *findShortestTrack* (from class *Playlist*) and *toString* (from class *Track*) to print the shorter of the two *Track* objects that was created and associated with the *Playlist* object in the *exam* method.
9. Let the *Track* class implement the *Comparable* interface. The natural order of *Track* objects is defined as alphabetical order of *artist* (secondary of *songName*).

Figure 1: *Sample Lab Test Exercise*

Each group of students get a different assignment. In principle the assignments are identical (they are all instances of the same generic assignment), but the students does not know nor realize this. The similarity of the assignments is important for fairness as well as comparability of the students' results. A sample assignment is presented in figure 1.

At the beginning of the exam, the students get a sheet of paper with the assignment consisting of nine small progressive programming tasks, and then they start programming. To ease inspection, we tell the students to tile all editor windows on the screen during the test.

When the first three tasks are finished, the students must demonstrate what they have achieved for one of the TAs. The lecturer and the external examiner evaluate the process as well as the product of each student, i.e. the students behaviour as well as the quality of the programs they produce counts in the final grading.

The sample lab test exercise in figure 1 is about tracks and playlists; the other exercises are about luggage and flights, employees and departments, side effects and medicine, etc. Although the concepts modelled by the classes vary, the assignments have similar structure. Because of this similarity, it is very easy for the lecturer and the external examiner without too much effort to evaluate the achievements of each student.

2.6 Apprenticeship Inspired Pedagogy

The course utilizes an apprenticeship-based pedagogy where students are exposed to how an expert programmer works. This is implemented by the lecturer behaving as a professional programmer (the master). For more information see [11].

The master reflects and thinks aloud of the particular action, maing them visible and as a source of identification [19]. As such, the apprentice (student) learns from observing the master (teacher) performing the actions embedded in the profession (e.g. coding, testing, etc).

3. RESEARCH METHOD

This paragraph discusses the methodology utilized in identifying the predictors of success for the model based CS 1 course described in the previous section. Section 3.1 outlines the research questions to be studied. Section 3.2 provides details on the subjects involved in the study. Section 3.3 describes the data and how it was provided, while Section 3.4 presents the manner in which data were collected and calculated.

3.1 Research Questions

In our current research, we look for potential success indicators that are statistically significant in predicting students' success when undertaking a model-driven introductory programming course. The factors are motivated by previous research in the field [13, 17, 27, 30].

1. What is the relationship of mathematical ability to model-based CS1?
2. What is the relationship of gender to model-based CS1?
3. What is the relationship of major/intended major to model-based CS1?
4. What is the relationship of course work to model-based CS1?
5. What is the relationship of years at the university to model-based CS1?
6. What is the relationship of the team to model-based CS1?

Due to technical problems, we did not collect data on the students feeling about the course, motivation for the course etc. Due to a technical problem, we were not able to use information about the students' previous programming experience.

3.2 Subjects

The subjects studied in this paper were students enrolled at the course Introduction to Programming at the University of Aarhus, Denmark, during the fall of 2004. Only data from students taking the course for the first time were used; to exclude the possibility of an extended practice effect, we decided to exclude from our investigations the students who followed the course for the second or third time.

3.3 Data

Several different data sources were used in this study. Information comes from the administrative system at the university (gender, enrollment date, major), the course web-site (team number), the teaching assistants (the score of the different lab-assignments), the final exam and the authors (the score in the exam) and a questionnaire (the math score for their high school exam).

Mathematical ability. The students score from their high school exam is used as an indicator of the students' mathematical abilities. The high schools in Denmark offer different levels of mathematical exams (A, B, and C where C is the lowest level). The students are required to have a high school math exam at the A level in order to take the introductory programming course. However, three students did not have the required A level but a B level. In our analysis, we observe that these three persons are outliers very far from the normal distribution, so they are excluded from the analysis. The students themselves in a questionnaire gave the score after the exam. A few students did not answer the questionnaire; they are also excluded from the analysis.

Course work. During the course, the students are required to complete five out of six weekly exercises in order to participate in the final exam. The teaching assistants evaluate the exercises and the score for each exercise is encoded as one of the numbers 1, 2, or 4. The interpretation of the encoding is:

Value	*Meaning*
1	Perfect, no significant errors
2	OK, small errors
4	Not accepted/Not handed in

Table 1: The scores for the weekly exercises

In case a student got a "4", he had the possibility of resubmitting the exercise once.

We have used the sum of these scores as a description of the students work during the course. We have excluded from our analysis the students who were not allowed to take the final exam.

Final exam score. The final exam is a practical test as described in section 2. The official result of the exam is a binary grading (pass or fail). In order for this research to be able to analyse the results at a finer grain, one author has post-marked all the students' solutions. The result of the more fine-grained marking is a grade in the interval [00...13] (see [10]). In order to pass an exam, a student needs a grade of 6 or more.

The official description of the grades is [10]:

13: Is given for the exceptionally independent and excellent performance.

11: Is given for the independent and excellent performance.

10: Is given for the excellent but not particularly independent performance.

9: Is given for the good performance, a little above average.

8: Is given for the average performance.

7: Is given for the mediocre performance, slightly below average.

6: Is given for the just acceptable performance.

5: Is given for the hesitant and not satisfactory performance.

03: Is given for the very hesitant, very insufficient and unsatisfactory performance.

00: Is given for the completely unacceptable performance.

The results of the post-marking is equivalent to the official results of the exam in the sense that all the students who passed the exam got a grade of six or more and the students who failed the exam got a grade of five or less. In order to ensure that the marking was fair, the co-author marked ten randomly selected answers. The results were identical.

In all the statistical tests, the result of the marking is used as the indicator of success—higher grade means more success.

3.4 Statistical Analysis

In order to test the hypotheses a covariance analysis is used. The analysis shows which (if any) of the independent variables that are correlated with the exam result.

The goal is furthermore to find how much impact (if any) the variables have on the result of the examination. One way to obtain this is to use a multiple regression analysis based on an as simple as possible model using the variables in question and the relevant interaction variables (i.e. combination of the variables).

In order to test the multiple regression model normally six prerequisites need to be fulfilled:

1. Linearity
2. Normal distribution
3. Homoscedasticity – the conditional distribution of Y has constant standard deviation throughout the range of values of the explanatory variables.
4. No collinearity – two or more variables have a strong linear relationship (i.e. explains the same).
5. No problematic outliers – an observation falls far from the rest of the data and the mean is highly influenced.
6. No autocorrelation - some observations are dependent.

Being population data, the requirements are not as important as if they were test samples. The team and the intended major have a correlation. This is to be expected since the teams are made up mostly of students with the same intended major. Team is therefore excluded from the analysis. The data fail on the test for normal distribution, but the test for 3, 4, and 5 is fine. Test 6 is only relevant for time series data. It is therefore possible to use multiple regression analysis in order both to check the hypotheses and furthermore to evaluate the impact the selected factors have on the actual exam result.

We start by running the complete multiple regression model with all variables including all the interaction variables. We find that the model explains 36.1% of the variation in the dependent variable at a 95% confidence interval. In order to meet the criteria of parsimony we compare the complete model with all interaction variables with a simple model i.e. a model without interaction variables. We find that we lose 34.4% explanation power since the simple model only explains 23.6% of the variation in the dependent variable. Because of the severe loss of explanation power in the simple model, we cannot ignore the model with all interaction variables. We want all variables in the model to be significant; this leads us to eliminate one by one all insignificant variables at a 95% confidence interval according to the hierarchical principle; we end up with a reduced model that explains 24.2% of the variance.

In the following, a 95% confidence interval is used to test the hypotheses (i.e. the probability of the hypotheses being true is 95%).

The analysis of the data is performed in SPSS version 13.0. The following variables are used:

Name	Description
GRADE	The result of the programming exam. Integer value from 00 – 13.
MATH	The score from the high school math exam. Integer value from 00 – 13.
COURSEWORK	The results of the assignments during the course. Integer value from 0-8. The variable is translated in the following way:

Sum of the results of the weekly assignments	*Value*
6	8
7	7
8	6
9	5
10	4
11	3
12	2
13	1
14	0

For an explanation of the results of the weekly assignments, see Table 1. If the sum of the assignments is 6, the student has handed in six perfect answers. If the value is 14 the student has handed in five acceptable assignments and one not acceptable/not handed in.

STUDYAGE	The number of years the student has been enrolled at the university. Integer value from 0 – 20. Students enrolled in 1984 or earlier were coded as 20.
COMPSCIENCE	The student intends to major in computer science (1=intended major in computer science, 0 otherwise).
GEOLOGY	The student intends to major in geology (1=intended major in geology, 0 otherwise).
MATHEMATICS	The student intends to major in math (1=intended major in math, 0 otherwise).
NANOSCIENCE	The student intends to major in nanoscience (1=intended major in nonoscience, 0 otherwise).
SEX	1= female, 0=male.

Table 2: *Description of the variables*

4. RESULTS

In this section the result of the multiple regressions is given.

4.1 Non significant variables

The variables NANOSCIENCE, MATHEMATIS, GEOLOGY, COMPSICENCE, SEX and STUDYAGE were not significant with respect to explaining the exam result using a 95% confidence interval. This was also the case with the interaction variables.

4.2 Multiple regression formula

The result of the regression analysis is presented in Table 3. The derived regression formula is:

```
GRADE =    1.118 +
           0.589*MATH +
           0.341*COURSEWORK
```

Variable	Unstandardised coefficients		Significance
	B	Std. Error	
COURSEWORK	0.341	0.097	0.000
MATH	0.589	0.107	0.001

Table 3: *Coefficients of the regression analysis*

The multiple regression formula (The reduced model with just two variables) explains 24.2 % of the variation of the exam grades. As described above the model with all the interaction variables explains 36.1% of the variation. The loss of explanation power in the reduced formula is 32.69%.

In order to find the importance of the different variables we have calculated the squared partial correlation coefficients (r^2). These describe the impact of one of the variables when the other variables are held fixed; in other words the amount of the variation of the exam grade that one of the variables is responsible for.

	r	r^2
COURSEWORK	0.264	0.069696 = 7 %
MATH	0.393	0.154449 = 15,4 %

Table 4: *Partial correlation coefficients*

In order to get a model that explains more of the variation of the exam grades we have tested the complete model using a 90% confidence interval for the individual variables. The reason is that it is population data. This gives the following formula:

```
GRADE =    -13.58 + 2.575*COURSEWORK +
           1.856*MATH + 3.564*COMPSCIENCE +
           6.668*GEOLOGY -
           0.673*MATH*GEOLOGY+
           0.064*MATH*STUDYAGE-
           0.192*COURSEWORK*MATH -
           0.515*COURSEWORK*COMPSCIENCE -
           0.111*COURSEWORK*STUDYAGE.
```

This formula accounts for 29.9% of the variation of the exam grade. In order to be compatible with the references, we will only discuss the model with a 95% confidence interval for the individual variables.

4.3 The hypotheses

In the following, we will discuss the research questions.

4.3.1 *Mathematical ability*
In the multiple regression formula, we can see that the math score from high school has a positive impact on the exam grade. We therefore accept the hypothesis that there is a positive correlation between the final exam score and the grade from the math exam in high school (95% confidence interval).

The squared partial correlation coefficient in the multiple regression was 15.4% saying that math grade alone accounts for over 15% of the variance of the final grade. This is almost the same that Leeper & Silver [17] found in their analysis; they found that math accounted for 14.3% of the variation.

4.3.2 *Gender*
The variables SEX was not significant, neither at the 95% confidence interval nor at the 90% confidence interval. We can therefore not accept the hypothesis that gender has an impact on the exam score. This corresponds with the findings of Ventura [28] "The tests fail to reveal any gender bias for course success." (p. 98), and the findings of [22].

4.3.3 *Major/intended major*
Neither of the variables indicating the intended major of the students were significant at the 95% confidence interval. This implies that we must reject the hypothesis of a positive impact of majoring in computer science. In the less accurate model, where the variables only were significant at the 90% confidence interval, the variables COMPSCIENCE and GEOLOGY were significant. At this level we can accept the hypothesis of a positive impact of majoring in computer science (it accounts for 3,6% of the variance), but since the variable GEOLOGY is significant but the variables NANOSCIENCE and MATHEMATICS are not, we can not say anything about the students not majoring in computer science. The finding corresponds with the findings in [27].

4.3.4 *Course work*
From the multiple regression formula, we can see that the variable COURSEWORK is significant at the 95% confidence interval and it has a positive impact on the exam grade. We can therefore accept the hypothesis that students who work harder get better grades.

The squared partial correlation coefficient in the linear regression was 7.0% indicating that course work alone accounts for 7% of the variance of the final grade; only half the impact of the math grade from high school.

4.3.5 *Study age*
The variable STUDYAGE was not significant at the 95 % confidence interval. We must therefore reject the hypothesis that there is a correlation between how many years the students have spend at the university and the result of the introductory programming course. Using a 90 % confidence interval, the variable is not significant in itself but in combination with the math grade, it has a positive impact; with course work, it has a negative impact. These two combinations of variables accounts for 2% of the variation each, but this is only at the 90% level.

4.3.6 *Team*
There is an a priori correlation between team and intended major because of the way students are allocated to teams, so the variable team was excluded from the model. Since intended major is not significant, the same is true for team.

5. Discussion
In this section, we discuss four aspects of our investigation: the explanation power of the variables, the impact of our findings on teaching, limits of what to conclude from the available data, and the variety of the notion "objects-first".

5.1 Explanation power of variables
The regression formula presented in section 4.2 accounts for 24.2% of the variation of the exam grade. One way to interpret this is that there is 75.8% not accounted for by these variables, so we cannot predict the actual grade from the two variables. This is the same conclusion that Leeper & Silver [17] reached; they used the regression formula on the students in next year's course and found they were only correct for 39 out of 106 students. On the other hand, we have only used two variables and using these, we can explain 24.2% of the variation of the exam grade – quite a large portion with only two variables. The variables considered here definitely have a large impact on the result of the exam.

5.2 Impact on teaching
The two variables we have found to be significant are math grade from high school and course work. The math grade counts for 2/3 of the explanation power of the two variables but unfortunately the students cannot improve it. Course work counts for 1/3 of the explanation power of the two variables, but opposite to the math grade, course work is improvable in the course and therefore interesting when designing the pedagogy of the course.

The significance of the course work variable indicates, not surprisingly, that students who follow the pace of the course performs better at the final exam. We discuss this aspect further in section 6 on future work.

5.3 Limits of conclusions
Prediction of success is difficult. Ventura [27] reached the conclusion that math score was not a success factor, we have found it to be!

This difference, of course, is a result of the origin of the data. Ventura's [27] and our data come from two different implementations of an objects-first CS1 programming course; we can only draw conclusions for each particular implementation, we cannot draw conclusions about success factors for objects-first programming courses in general. In order to answer the more general question of success factors of an object-first CS1 course we need data from various different implementations of this teaching strategy.

5.4 Objects-first
Objects-first is not a well-defined term. It seems that every CS1 teacher has his or her own interpretation of the term (e.g. 9, 15, 16, 24]. In [14] the description of objects-first is: "an objects-first approach that emphasizes early use of objects and object-oriented design" (p. 28). What does early mean, and what is meant by object-oriented design?

In [18] the author discuss nine myths about object-orientation and its pedagogy; one is that the phrase "objects first" is well.defined. The author writes: "No matter what your definition of objects first is, it is likely to be different from that of the person next to you." (p. 247), and "The phrases 'objects first' and 'objects early' are bandied about in a variety of contexts. When discussing a CSI course they are often used to convey the general idea that objects are discussed early in the course and established as a fundamental concept. Beyond that, however, these phrases seem to take on a variety of meanings, with important implications." (p. 246).

Because of the variety of interpretations of "objects-first", it is impossible to make conclusions about this approach in general.

6. FUTURE WORK

In the current research, we have investigated the relationship between the student's achievements in the final exam of the introductory programming course and mathematical prerequisites, gender, study program, student team, maturity, and the student's achievements in the mandatory weekly exercises during the course.

Identifying success factors is relevant, and has been done in many fields with many different hypotheses of success factors for education in specific fields and for education in general. Wang & Hertel [29] abstracted over more than 11.000 statistical findings in order to identify the most influential factors for learning. They found that "the students metacognitive processes that is, a student's capacity to plan, monitor, and, if necessary, re-plan learning strategies—had the most powerful effect on his or her learning." (p.75). Even though their research was based on students in primary and lower secondary schools, other research have found factors related to student aptitude or classroom management to be important as well in a university setting. For references, see [21].

The explanation power of the variables we have studied is rather small. However, more important is the fact that the most influential of the variables from our study, math grade from high school, is outside our control—we cannot do anything to improve it by changing the course design, and the students cannot do anything about it by changing their attitude in the CS1 course. We would like to identify success factors within our control, i.e. success factors that we can promote by changing aspects of our course design.

From our experience, we conjecture that other factors also are likely to be indicators of success than the ones investigated in the research reported in this paper. Numerous other factors might be success indicators in an introductory model-driven programming course, e.g. motivation, effort, power of abstraction, prior programming experience, social course context, emotional and social health, family background, ethnic background, financial situation, and computer literacy. In order to improve the learning situation, we would like to pursue those factors we believe to be dominant in predicting success and which we can do something about by changing our course design, and this rules out family background, ethnic background, financial situation and computing literacy.

Motivation. How motivated is the student? Presumably, a CS major is more motivated than a math major or chemistry major; and of course some CS students are more motivated than others.

Effort. How hard does the student work with the subject during the quarter/semester? Programming is a contact-sport, and the hard-working students are likely to perform vastly better than the less hard-working students are. In this research, we have used a simplified description of the effort the students puts in the course namely the result of the mandatory assignments.

Power of abstraction. We believe that the student's power of abstraction —the students ability to cope with abstract concepts and their detailed realization in a modern programming language which is a task spanning several orders of magnitude— plays a dominant role as indicator of success in any introductor programming course, also a model-driven course as ours.

Prior programming experience. All other things being equal, we expect prior programming experience to be an indicator of success. However, one often sees that students with prior programming experience rely too much on their prior experience and eventually find themselves (lost) far behind the students that approach their study with a more humble and hard-working attitude. This indicator has been shown in several studies [13, 30] to be an indicator of success, even though Ventura [28] could not confirm this.

Social course context. We believe the social context of the learning environment, i.e. the lecturer, the TA, and the fellow students, to be an indicator of success; however, it is probably a more moderate success indicator than the other four mentioned above.

Emotional and social health. [21] found that "both emotional and social health factors related to student performance and retention" (p24). In their study they have used a wide range of tests to determine college students emotional and social health. It would be interesting to see if these factors have the same impact on students participating in a model-based programming course.

It is not a trivial task to measure the parameters mentioned above; consequently, a major part of the indicated future work will be to identify trustworthy techniques of establishing quantitative measures of these parameters.

In section 5.3 we concluded: "In order to answer the more general question of success factors of the object-first CS1 course we need data from various different implementations of this teaching strategy." Therefore, we would like to extend the investigation to other institutions (other teachers, other interpretations of objects-first, etc.).

7. CONCLUSIONS

We have studied eight potential indicators of success for a model-driven CS1 course at university level: math grade from high school, course work, study age, major in CS, major in math, major in geology, major in nano science, and gender.

We have explained our specific interpretation of objects-first by presenting a detailed description of the course design including goal, form, content, exam, and pedagogy.

We have presented our research method including research question, data, statistical method (multiple regression analysis).

Of the eight potential indicators of success, we have found only two to be significant at a 95% confidence interval: math grade from high school and course work. The two significant indicators explain 24.2% of the variation of the exam grade. Math is the more dominant of the two, it accounts for 2/3 of the variation. The result concerning math grade contradicts the findings of Ventura [27].

We have discussed four aspects of our research:

1. The explanation power of the variables: the variables considered here definitely have a large impact on the result of the exam.
2. The impact of our findings on teaching: the significance of the course work variable indicates, not surprisingly, that students

who follow the pace of the course performs better at the final exam.

3. Limits of what to conclude from the available data: data from various different implementations of this teaching strategy is needed in order to answer the more general question of success factors of an object-first CS1 course.
4. The variety of the notion “objects-first”: because of the variety of interpretations of “objects-first”, it is impossible to make conclusions about this approach in general.

Further work need to be done in order to make generalizable results on the success factors for objects-first programming; we suggest six potential indicators of success that we believe to be dominant in predicting success and which we can do something about by changing our course design.

8. ACKNOWLEDGEMENT

We would like to thank Louise Heltborg for her help in the statistical analysis and Lisbeth Petersen for her help in collecting the data. We would like to thank all the participants in the course for taking part in our investigations.

9. REFERENCES

[1] Bennedsen, J. & Caspersen, M.E.(2004). Programming in Context – A Model-First Approach to CS1, Proceedings of the thirty-fifth SIGCSE Technical Symposium on Computer Science Education, Norfolk, Virginia, 2004, pp. 477-481.

[2] Bennedsen J. & Caspersen, M.E (2005). Revealing the Programming Process. Proceedings of the thirty-sixth SIGCSE Technical Symposium on Computer Science Education, St. Louis, Missouri, 2005. pp.186-190.

[3] Bergin, J. 14 Pedagogical Patterns. Available on-line at “http://csis.pace.edu/~bergin/PedPat1.3.html”. Last accessed May 13 2005.

[4] Bergin, S & Reilly, R (2005) Programming: factors that influence success. SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education ,St. Louis, Missouri, USA. pp. 411--415.

[5] Brooks, J. H., & DuBois, D. L. (1995). Individual and environmental predictors of adjustment during the first year of college. Journal of College Student Development, 36, pp. 347-360.

[6] Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. Proceedings of the 6th annual conference on Innovation and technology in computer science education, 49-52.

[7] Caspersen, M.E. & Christensen, H.B. (2000). Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS1. Proceedings of the Fourth Australasian Computing Education Conference, ACE 2000 Melbourne, Australia, 2000, pp. 34-40.

[8] Caspersen, M.E. & Christensen, H.B.(2002) Frameworks in CS1 -- a Different Way of Introducing Event-driven Programming. In: Proceedings of the seventh Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2002, Aarhus, Denmark.

[9] Cooper, S., Dann, W.& Pausch, R. (2003) Teaching Objects-first In Introductory Computer Science SIGCSE’03 February 19-23, 2003, Reno, Nevada, USA. pp. 85 – 89.

[10] Exam score (2005). http://www.retsinfo.dk/_GETDOCM_/ ACCN/B19950051305-REGL (English translation can be found in the bottom) Last accessed April 30 2005.

[11] Fjuk, A., Berge, O., Bennedsen, J. & Caspersen, M. (2004). Learning Object-Orientation through ICT-mediated Apprenticeship. Procedings of the 4th IEEE International Conference on Advanced Learning Technologies, Joensuu, Finland.

[12] Foster, T. R. (1998). A comparative study of the study skills, self-concept, academic achievement and adjustment to college of freshman intercollegiate athletes and nonathletes. Dissertation Abstracts International Section A: Humanities and Social Sciences, 58(12-A), pp. 4565.

[13] Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? ACM SIGCSE Bulletin , 5th annual SIGCSE/SIGCUE conference on Innovation and technology in computer science education, 32(3). pp. 25-28.

[14] The Joint Task Force on Computing Curricula (IEEE Computer Society and Association for Computing Machinery). Computing Curricula 2001 (final report), December 2001. Available on-line at “http://www.computer.org/education/cc2001/final”. Last accessed May 13, 2005.

[15] Jones, R., Boyle, T. & Pickard, P. (2003) Objectworld: Helping Novice Programmers to Succed through a Graphical Objects-first Approach. Proceedings of 4th Annual LTSN-ICS Conference, NUI Galway, pp. 111 – 114.

[16] Kölling, M. & Rosenberg, M. (2001) Guidelines for teaching object orientation with Java, ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education, pp. 33 – 36.

[17] Leeper, R. R., & Silver, J. L. (1982). Predicting success in a first programming course. Technical Symposium on Computer Science Education, Proceedings of the thirteenth SIGCSE technical symposium on Computer science education, Indianapolis, Indiana, United States. pp: 147 – 150.

[18] Lewis, J. (2000) Myths about object-orientation and its pedagogy, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, pp. 245-249.

[19] Nielsen, K., Kvale, S. (1997). Current issues of apprenticeship”. Nordisk Pedagogik, Vol 17, pp. 130-139.

[20] Nowaczyk, R. H. (1983). Cognitive skills needed in computer programming. Paper presented at the Annual Meeting of the Southeastern Psychological Association, Atlanta, Georgia.

[21] Pritchard, M. E. & Wilson G S. (2003). Using Emotional and Social Factors to Predict Student Success, Journal of College Student Development, Vol 44(1).

[22] Rountree, N. Rountree, J. and Robins, A (2002).Predictors of success and failure in a CS1 course. SIGCSE Bulletin, vol 34(4) pp. 121—124.

[23] Sanders, R. T., Jr. (1998). Intellectual and psychosocial predictors of success in the college transition: A multiethnic

study of freshman students on a predominantly White campus. Dissertation Abstracts International Section B: The Sciences and Engineering, 58(10-B), pp. 5655.

[24] Schmolitzky, A. (2004) Objects first, interfaces next" or interfaces before inheritance, Educators symposium, OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications pp. 64 – 67.

[25] Szulecka, T. K., Springett, N. R., & de Pauw, K. W. (1987). General health, psychiatric vulnerability and withdrawal from university in first-year undergraduates. British Journal of Guidance & Counseling Special Issue: Counseling and health, 15, pp. 82-91.

[26] Ting, S. R., & Robinson, T. L. (1998). First-year academic success: A prediction combining cognitive and psychosocial variables for Caucasian and African American students. Journal of College Student Development, 39, pp. 599-610.

[27] Ventura, P. R.. (2003). On the Origins of Programmers: Identifying Predictors of Success for an Objects First CS1", PhD. dissertation, The State University of New York at Buffalo, 2003.

[28] Ventura, P. R. & Ramamurthy, B. (2004). Wanted: CS1 Students. No Experience Required. ACM SIGCSE Bulletin , Proceedings of the 35th SIGCSE technical symposium on Computer science education, Volume 36(1) pp. 240 – 244.

[29] Wang, M. C. & Haertel, G. D. (1993). What helps students learn? Educational Leadership; Dec93/Jan94, Vol. 51 Issue 4, pp. 74-79.

[30] Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. ACM SIGCSE Bulletin , Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education, 33(1), pp. 184-188

13 Abstraction Ability as an Indicator of Success?

The paper *Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?* presented in this chapter has been published as a journal paper [Bennedsen et al. 2006a].

[Bennedsen et al. 2006a] Bennedsen, J. and Caspersen, M.E., "Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?", *SIGCSE Bulletin*, vol. 38, 2, pp. 39-43, 2006.

Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming?

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

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34
DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

Abstract
Computer science educators generally agree that abstract thinking is a crucial component for learning computer science in general and programming in particular. We report on a study to confirm the hypothesis that general abstraction ability has a positive impact on programming ability. Abstraction ability is operationalized as stages of cognitive development (for which validated tests exist). Programming ability is operationalized as grade in the final assessment of a model-based objects-first CS1. The validity of the operationalizations is discussed. Surprisingly, our study shows that there is no correlation between stage of cognitive development (abstraction ability) and final grade in CS1 (programming ability). Possible explanations are identified.

Keywords: CS1, success factors, abstraction, model-based programming, objects-first.

1. Introduction

A substantial amount of research has been conducted in order to identify variables that are predictors of success of students aiming for a university degree. Investigated variables encompass among other things gender [4, 17, 24], the educational level of parents [20] and ACT/SAT scores [4, 14]. The variables represent scientific factors (e.g. math score) or unbiased factors (e.g. gender). However, these variables only account for a fraction of the variation of student performance.

Research on success factors has been conducted both in the general context of education, within computer science, and in the more specific area of introductory programming [4, 6, 9, 14]. Also in the area of introductory object-oriented programming there has been research trying to establish general factors to predict success or failure of particular students. Especially the work of Ventura [21] focus on a systematic evaluation of hypothesis related to success factors of an introductory programming course using an objects-first and graphics early approach [22, p.241]. The results are also documented in [23].

We are specifically interested in abstraction ability as an indicator of success for learning programming. Most computer science teachers find abstract thinking to be a core competence in programming, but to our knowledge no research has been conducted to verify whether abstraction ability is actually a predictor of success of an introductory programming course using an objects-first strategy [3].

2. Abstraction Ability and Programming

Many computer science educators argue that abstraction is a core competence [2, 13, 15, 16, 19].

Nguyen & Wong [15] claim that it is difficult for many students to learn abstract thinking; at the same time they claim abstract thinking to be a crucial component for learning computer science in general and programming in particular. The authors describe an objects-first-with-design-patterns approach to CS1 with a strong focus on abstract thinking and development of the students' abstractive skills.

In [16] the authors argue that abstraction is a fundamental concept in programming in general and in object-oriented programming in particular. The authors describe a three-level ordering of abstraction cognitive activities that the students employ in their solution to a given problem: 1) defining a concrete class, 2) defining an abstract class with attributes only, 3) defining an abstract class also including methods, and 4) defining an abstract class also including abstract methods). An analysis of the students' responses to a test reveals that only 13% apply the highest level of abstraction cognitive activities (level 4) while 65% solve the problem at the lowest level of abstraction cognitive processes. The authors conclude that the major cited advantages of object-orientation are precisely the same issues that make object-orientation difficult for students.

2.1 Hypothesis

Clearly, abstraction and abstract thinking are fundamental concepts in computer science and key components of learning programming. For programming education (and CS education in general) it is therefore mandatory to explicitly aim at the development of the students' abstractive skills. But furthermore we anticipate general abstractive skills — abstraction ability— to be an indicator of success for learning programming. Our hypothesis is therefore:

General abstraction ability has a positive impact on programming ability.

2.2 Abstraction Ability as Stages of Cognitive Development

To operationalize the first part of our hypothesis we need to define what we mean by abstraction ability and how it can be measured. Or-Bach & Lavy [16] define abstraction ability in terms of object-oriented programming. However, abstraction ability is a much more general skill often defined as part of the cognitive development stage of a person [11]. Our approximation of abstraction ability is based on Adey & Shayer's theory of cognitive development [1, 18]; this theory is a refinement of Inhelder & Piaget's stage theory [11]. Adey & Shayer define eight stages of cognitive development of pupils [1, p. 30] as shown in Table 1.

Table 1: *Cognitive development stages*

1	Pre-operational
2A	Early concrete
2A/2B	Mid concrete
2B	Late concrete
2B*	Concrete generalization
3A	Early formal
3A/3B	Mature formal
3B	Formal generalization

Adey & Shayer based their stages of cognitive development on a very large research project, CASE, aimed at finding the cognitive development stages of pupils in secondary school [1, p.78 ff]. The research showed a different result than the direct connection between age and development stage originally proposed by Piaget. One of the most important results was that only ~30% of the pupils follow the development expected by Piaget.

Based on [11], Adey and Shayer describe what they call "reasoning patterns of formal operations" and group the eight patterns in three groups: Handling of variables, relationships between variables and formal methods. See [1, pp.17-25] for a more exhaustive description. A person can of course be at a higher development stage in one of these reasoning patterns, but "one would not find an individual competently fluent with one or two of the reasoning patterns who would not, with very little experience, become fluent with them all" [1, p.17].

Shayer and Adey have used the eight stages for pupils in the age range of 5 to 16; we intend to use it on students in the age range of 18 to 22. Shayer and Adey found that at the age of 16, 30% of the pupils were at stage 3A and only approximately 10% at stage 3B. Furthermore they found that the curve describing the progression of stages was very flat at that age [1, p.40].

We use Adey & Shayer's stage model of cognitive development to characterize the students' abstraction ability. To measure abstraction ability defined in this way, we use a reasoning ability test developed by Piaget and refined by Adey & Shayer for testing at the higher end of the stage model.

2.3 Programming Ability as Final Grade in CS1

To operationalize the second part of our hypothesis we need to define what we mean by programming ability and how it can be measured. In this research we use the results from the final exam of the introductory programming course as an indicator of the students' programming ability. For a more thorough description of the course, see [3].

2.4 A Word on the Operationalization

The hypothesis that general abstraction ability has positive impact on programming ability is operationalized in two steps; abstraction ability is operationalized as cognitive development and programming ability is operationalized as final grade in CS1 as illustrated in Figure 1. Both of these operationalizations are questionable. We discuss this aspect in the section on future work.

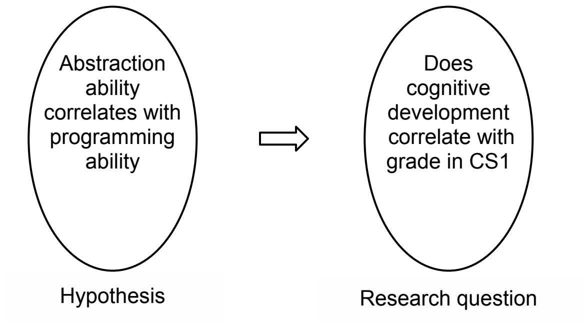

Abstraction ability correlates with programming ability
Does cognitive development correlate with grade in CS1
Hypothesis
Research question

Figure 1: *Operationalization of hypothesis*

3. Research

This section describes the research questions, the data and the statistical analysis used in this work.

3.1 The research questions

Our hypothesis is that there is a positive correlation between the stage of a student's cognitive development (measured as reasoning ability) and the students programming ability (measured as final grade in CS1).

Many reports that math is an indicator of success in programming [4, 9, 14]. Our interpretation of this fact is that it is not specific mathematical competencies (e.g. calculus and algebra) that the students need, but rather the more general notion of abstraction ability required to do math that is needed.

To verify our interpretation, we propose a supplementary research question on the correlation between abstraction ability and mathematical competence. Our two research questions are therefore:

1. *Is there a positive correlation between the stage of cognitive development and the students' results in model-based introductory programming?*
2. *Is math an indicator of the cognitive development stage?*

3.2 The Test

Shayer & Adey have developed several tests to determine the students' cognitive stages. These test focus on several of the reasoning patterns, but because "the students with very little experience, become fluent with them all" we find it sufficient to use only one test. We use the so called "pendulum test"; a test that has been used for a long time to test young persons' understanding of the laws of the physical world [7]. Shayer and Adey argues that the pendulum test is particular focused on testing the cognitive development stages from 2B to 3B [1, p.30], the span of cognitive stages we find relevant for our target group.

The students volunteered to participate in the test. It was given to them in a lecture hall, and they were all informed that the outcome of the test would not be exposed to the lecturer before the exam.

3.3 The Students

The students in this research all study at the Faculty of Science at University of Aarhus in Denmark. They all follow an introductory programming course as a mandatory part of their study programme. The course constitutes the first half of a traditional CS1 course. The course runs for seven weeks. Every week there are four lecture hours, two lab hours and two class hours with a teaching assistant (TA). Besides scheduled hours, the students are supposed to work approximately seven hours per week in study groups or on their own. A week after the course there is a practical exam with a binary pass/fail grading. For a more detailed description of the final exam see [5].

In the fall of 2005 there were 263 students from a variety of study programmes, e.g. computer science, mathematics, mathematical economy, multimedia, geology, nano science, etc. Approximately 40 % of the students are enrolled for a major in computer science and they are the only group to continue with the second half of CS1. The rest of the students proceed to other programming courses related to their fields (e.g. multimedia programming, scientific computing) if they proceed with programming at all.

The goal is that the student learns the foundation for systematic construction of simple programs and through this obtains knowledge about the role of conceptual modeling in object-oriented programming. Furthermore, it is the goal that the student becomes familiar with a modern programming language, fundamental programming language concepts, and selected class libraries. For further details on the structure and contents of the course see [3].

3.4 Data

Information about the score of final exam comes from the administrative system of the university.

Programming score. The final exam is a practical programming test. The official result of the exam is a binary grading (pass or fail). To allow for a more fine-grained analysis of the results, the students' solutions were post-marked on an A-F scale. To validate the result of the post-marking, the post-marking was compared to the official result of the exam in the sense that all the students who passed the exam got a grade of E or more. Also, the result of the post-marking was checked by a control marking of twenty randomly selected answers. The marking and the control marking agreed.

Math score. The students' math score from high school was used as an indicator of the students' mathematical abilities. The students themselves gave their math score in a questionnaire. A few students did not answer the questionnaire; these students were excluded from the analysis.

3.5 Statistical analysis

We have used a Pearson correlation coefficient test to find if there is a significant correlation between the result of the exam and the cognitive development stage and math score. Of the 263 students who took the final exam, 145 participated in the pendulum test. They are representative of the overall student group with respect to mathematical skills, gender and intended major.

4. Results

In this section we describe the analysis providing the answers to the two research questions.

4.1 No Correlation Between Cognitive Development and Programming Ability

As described above we have calculated Pearson correlation between cognitive development and programming ability (Table 2). The coefficient, R, is 0.276 which indicates a very weak correlation (a value of at least 0.3 indicates correlation). The significance, P, is less than 0.001.

This is a rather unexpected result, since most computer science educators seem to agree that abstraction ability – and thereby cognitive development – is a core competence in programming. Our research cannot demonstrate a correlation between the stage of cognitive development and the students' results in a model-based introductory programming course.

Table 2: Correlation between cognitive development and programming ability

Pearson correlation test	
R	0.276409
R^2	0.076402
P	0.000764
Observations	145

Cafolla [10] reports that the stage of cognitive development accounts for 34 % of variation of the exam score. Cafolla's study is based upon students learning programming in BASIC. It seems unlikely that BASIC programming should require a higher degree of cognitive development than object-oriented programming; we need to investigate this more thoroughly.

4.2 No Correlation between Math and Cognitive Development

We have also calculated Pearson correlation between the score of the programming exam and the math score from high school. The exam in high school is a nation vide test in two parts: a written and an oral test. The written test is administered by the Ministry of Education. We have used the average of the two exam scores as the math score. Of the 143 students participating in the pendulum test, 128 provided their math score.

As can bee seen from table 3, there is hardly any correlation between the students' mathematical ability and their cognitive stage. Again this comes as a surprise as the expected result was a strong correlation between math and formal cognitive development. The result contradicts earlier findings, summarized in [12, p.260].

Table 3: Correlation between stage of cognitive development and mathematical ability

Pearson correlation test	
R	0.186781261
R^2	0.034887239
P	0.034766
Observations	128

The correlation that others have found between math and success in programming is not contradicted by our data (R= 0.302191, p=0.000555). From our experiment we must conclude that math is not just another way of expressing the cognitive development stage and that the correlation between math and success in programming must be related to other aspects of math.

5. Conclusion and Future Work

The result of this study is most surprising. From the outset we were certain that students at a higher stage of cognitive development would get higher scores in the final exam of the introductory programming course. It is not so!

There can be several explanations to this. In this programming course coding is prioritized over design. The cognitive requirements are therefore relatively low, and apparently there are other factors that influence the students' success. We will look into this in future work.

Another potential explanation is the concrete instrument used to assess the cognitive stage: the pendulum test. The pendulum test measures the student's ability to control independent variables in a reasoning task. It could be that this particular competence is not prominent in the course.

Finally, of course, it is questionable to which extend the result of the final exam is a reasonable measure of a student's ability to learn programming.

Acknowledgement

It is a pleasure to acknowledge Jens Holbech for his help regarding measurement of cognitive stage (abstraction ability). Also, we would like to thank all the students who volunteered for this study.

References

[1] Adey, P and Shayer, M. Really raising standards: cognitive intervention and academic achievement, *Routledge*, London, England, 1994.

[2] Alphonce, C. and Ventura, P. Object Orientation in CS1-CS2 by Design, *Proceedings of the 7th Annual Conference on innovation and Technology in Computer Science Education*, Aarhus, Denmark, 2002, 70-74.

[3] Bennedsen, J. & Caspersen, M.E. Programming in Context – A Model-First Approach to CS1, *Proceedings of the thirty-fifth SIGCSE Technical Symposium on Computer Science Education*, Norfolk, USA, 2004, 477-481.

[4] Bennedsen, J & Caspersen, M. E. An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course, *Proceedings of ICER 2005 The First International Computing Education Research Workshop*, 2005, Seattle, USA, 155-163.

[5] Bennedsen, J. & Caspersen, M.E. Assessing Process and Product – A Practical Lab Exam for an Introductory Programming Course, Submitted for *36th Annual Frontiers in Education Conference*, San Diego, USA, 2006.

[6] Bergin, S & Reilly, R. Programming: Factors that Influence Success, *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, USA, 2005, 411-415.

[7] Bond, T. B. Piaget and the Pendulum, *Science and Education*, 13, 2004, 389-399.

[8] Boyer, S. P., & Sedlacek, W. E. Non-Cognitive Predictors of Academic Success for International Students: A Longitudinal Study, *Journal of College Student Development*, 29, 1988, 218-223.

[9] Byrne, P., & Lyons, G. The Effect of Student Attributes on Success in Programming, *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, 2001, 49-52.

[10] Cafolla, R. Piagetian Formal Operations and other Cognitive Correlates of Achivement in Computer Programming, *Journal of Educational Technology Systems*, 16(1), 1987-88, 45-55.

[11] Inhelder, B. & Piaget, J. (1955) *De la logique de l'enfant à la logique de l'adolescent: Essai sur la construction des structures opératoires formelles*. Paris: Presses Universitaires de France. Translated by Anne Parsons and Stanley Milgram as *The growth of logical thinking from childhood to adolescence: An essay on the construction of formal operational structures,* New York: Basic Books, 1958.

[12] Iqbal, H.M. and Shayer, M. Accelerating the Development of Formal Thinking in Pakistan Secondary School Students: Achievement Effects and Professional Development Issues, *Journal of Research in Science Teaching*, 37 (3), 2000, 259-274.

[13] Kurtz, B. L. Investigating the Relationship Between the Development of Abstract Reasoning and Performance in an Introductory Programming Class, *Proceedings of the 11th SIGCSE Technical Symposium on Computer Science Education*, Kansas City, USA, 1980, 110-117.

[14] Leeper, R. R., & Silver, J. L. Predicting Success in a First Programming Course, *Proceedings of the 13th SIGCSE Technical Symposium on Computer Science Education*, Indianapolis, USA, 1982, 147 – 150.

[15] Nguyen, D. & Wong, S. OOP in Introductory CS: Better Students Through Abstraction, *Proceedings of the fifth Workshop on Pedagogies and Tools for Assimilating Object-Oriented Concepts*, OOPSLA 2001.

[16] Or-Bach, R. and Lavy, I. Cognitive Activities of Abstraction in Object Orientation: An Empirical Study. *SIGCSE Bulletin*, 36 (2), 2004, 82-86.

[17] Rountree, N. Rountree, J. and Robins, A. Predictors of Success and Failure in a CS1 Course. SIGCSE Bulletin, vol. 34 (4), 2002, 121-124.

[18] Shayer, M. and Adey, P. Towards a Science of Science Teaching, *Heinemann Educational Publishers*, Oxford, England, 1981.

[19] Sprague, P., & Schahczenski, C. Abstraction the Key to CS1. *J.Comput.Small Coll., 17* (3), 2002, 211-218.

[20] Ting, S. R., & Robinson, T. L. First-Year Academic Success: A Prediction Combining Cognitive and Psychosocial Variables for Caucasian and African American Students, *Journal of College Student Development*, 39, 1998, 599-610.

[21] Ventura, P. R. *On the Origins of Programmers: Identifying Predictors of Success for an Objects First CS1*, PhD. Dissertation, The State University of New York at Buffalo, 2003.

[22] Ventura, P. R. & Ramamurthy, B. Wanted: CS1 Students. No Experience Required, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, USA, 2004, 240-244.

[23] Ventura, P.R. Identifying Predictors of Success for an Objects-First CS1, *Journal of Computer Science Education*, 15 (3), 2005, 223-243.

[24] Wilson, B.C. A Study of Factors Promoting Success in Computer Science Including Gender Differences, *Journal of Computer Science Education*, 12 (1-2), 2002, 141-164.

Check out the new website of

CSAB

<www.csab.org>

14 Mental Models and Programming Aptitude

The paper *Mental Models and Programming Aptitude* presented in this chapter has been submitted for ITiCSE 2007 [Caspersen et al. 2007a].

[Caspersen et al. 2007a] Caspersen, M.E., Bennedsen, J. and Larsen, K.D., “Mental models and programming aptitude”, submitted for *ITiCSE '07: The 12th annual conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland, 2007.

Mental Models and Programming Aptitude

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34
DK-8200 Aarhus N, Denmark
mec@daimi.au.dk

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

Kasper Dalgaard Larsen
Department of Computer Science
University of Aarhus
Aabogade 34
DK-8200 Aarhus N, Denmark
larsen@daimi.au.dk

ABSTRACT
Predicting the success of students participating in introductory programming courses has been an active research area for more than 25 years. Until recently, no variables or tests have had any significant predictive power. However, Dehnadi and Bornat claim to have found a simple test for programming aptitude to cleanly separate programming sheep from non-programming goats. We briefly present their theory and test instrument.

We have repeated their test in our local context in order to verify and perhaps generalise their findings, but we could not show that the test predicts students' success in our introductory programming course.

Based on this failure of the test instrument, we discuss various explanations for our differing results and suggest a research method from which it may be possible to generalise local results in this area. Furthermore, we discuss and criticize Dehnadi and Bornat's programming aptitude test and devise alternative test instruments.

Categories and Subject Descriptors
K3.2 [**Computers & Education**]: Computer and Information Science Education – *computer science education, information systems education*.

General Terms
Experimentation, Human Factors.

Keywords
Objects-first, CS1, introductory programming, object-oriented programming, predictors of success.

1. INTRODUCTION
In a teaser email circulated in late 2005, shortly before the PPIG workshop in January 2006, Richard Bornat wrote: "We have a scientific breakthrough that we'd like to announce at your little PPIG. The breakthrough is that Saeed has a test which picks out, with 100% accuracy, those people who have a chance of learning to program and rejects, with 100% accuracy, those who have no chance. Don't believe it? Neither did I, at first, but it's true. And I'm not telling you, before the little PPIG, just how it's done. But of course I will tell you all there."

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE 2007, June 25–27, 2007, Dundee, Scotland.
Copyright 2007 ACM X-XXXXX-XX-X/XX/X.

We learned about the test in conjunction with the PPIG workshop in January 2006. Having searched for predictors of success for introductory programming courses, we were certainly intrigued by the promotion material, and we decided to try to verify Dehnadi and Bornat's findings.

This paper describes what we found. As has already been noted in the abstract, we have not been able to verify Dehnadi and Bornat's findings.

In the next section, we provide a brief overview of some of the research which aims at finding predictors of success for computer science studies at universities. In section 3, we describe the programming aptitude test developed at Middlesex University by Dehnadi and Bornat and their preliminary results. In sections 4 and 5, we present our research method and our findings, which we discuss in section 6. Section 7 provides a succinct conclusion.

2. RELATED WORK
There has been a substantial amount of research conducted to identify general variables that are predictors of the success of students aiming for a degree in computer science. Investigated variables encompass gender [21, 22], ACT/SAT scores [9], students' mathematical abilities [6, 19, 22], performance in prior courses [12], emotional factors [11], abstraction ability [3], and students' own beliefs [24]. Research has also been conducted in the more specific area of introductory programming [2, 5, 8, 10, 17, 18, 20].

Evans and Simkin [15] sum up the arguments given in many studies for performing this kind of study:

1. Discriminating among enrolment applicants
2. Advising students on majors
3. Identifying productive programmers
4. Identifying employees who might best profit from additional training
5. Improving computer classes for non-CIS majors
6. Determining the importance of oft-cited predictors of computer competency such as gender or math ability
7. Exploring the relationship between programming abilities and other cognitive reasoning processes

Dehnadi and Bornat [1, 13, 14] claim they have found a way to identify students who will not succeed in learning programming. Based on a test of 60 students, they claim "[w]e have found a test for programming aptitude, of which we give details. Remarkably, we can predict success or failure even before students have had any contact with any programming language, and with total accuracy." [14].

Our rationale for conducting this study is primarily to verify the claims made by Dehnadi and Bornat and to build up knowledge about factors that influence students' learning of programming.

3. TEST FOR PROGRAMMING APTITUDE

In this section, we describe the programming aptitude test developed at Middlesex University by Dehnadi and Bornat and their preliminary results as reported in [14].

Dehnadi and Bornat classified students according to their consistency in answering a set of similar questions. The overall hypothesis is that consistent students and consistent students only will be able to learn to program.

To determine consistency, Dehnadi and Bornat used a questionnaire with 12 small Java programs. Each program consists of two variable declarations and one, two, or three assignment statements; Figure 1 shows a sample.

5. Read the following statements and tick the box next to the correct answer in the next column.	**The new values of a and b are:**
	☐ `a = 30      b = 50`
	☐ `a = 10      b = 10`
	☐ `a = 20      b = 20`
	☐ `a = 10      b =  0`
`int  a = 10;`	☐ `a =  0      b = 20`
`int  b = 20;`	☐ `a = 30      b =  0`
	☐ `a = 40      b = 30`
`b = a;`	☐ `a =  0      b = 30`
`a = b;`	☐ `a = 20      b = 10`
	☐ `a = 30      b = 30`
	☐ `a = 10      b = 20`
	Any other values for a and b:
	`a =          b =`
	`a =          b =`
	`a =          b =`

Figure 1: *A sample question from Dehnadi and Bornat's questionnaire*

Dehnadi and Bornat have identified 11 different mental models which are captured by options in the questionnaire (along with the last option: *other*). The questionnaire contains 12 questions similar to the one in Figure 1, giving rise to a 12-tuple describing the mental models applied by a student (e.g. (m_7, m_3, ..., m_7)) where m_i represents a mental model. The 12-tuple is used to *assign* each student to one of three categories:

- The *consistent* group. The students who use the same mental model for most of the questions (irregardless of which model).
- The *inconsistent* group. The students who use varying mental models for the questions.
- The *blank* group. The students who refuse to answer the questions.

In [13], the authors write: "The consistent/inconsistent/blank assignment which is the basis of our preliminary result was rather subjective". In [13], the authors develop a more objective instrument for categorisation of the students—an instrument which we shall use in our investigation.

Dehnadi and Bornat found that 44% of their students belong to the consistent group, and 39% belong to the inconsistent group; 8% left the questionnaire blank (the remaining 9% are missing).

In [14], the authors conclude that the test, although not perfect, is the first test to be able to claim any degree of success:

> "[Our analysis] shows that the first administration of Dehnadi's test reliably separated the consistent group, who almost all scored 50 or above, from the rest, who almost all scored below 50, with only 4 out of 27 false positives in the consistent group and 9 out of 34 false negatives in the rest [...]. Clearly, Dehnadi's test is not a perfect divider of programming sheep from non-programming goats. Nevertheless, if it [was] used as an admissions barrier, and only those who scored consistently were admitted, the pass/fail statistics would be transformed. In the total population 32 out of 61 (52%) failed; in the first-test consistent group only 6 out of 27 (22%). We believe that we can claim that we have a predictive test which can be taken prior to the course to determine, with a very high degree of accuracy, which students will be successful. This is, so far as we are aware, the first test to be able to claim any degree of predictive success."

It is indeed very interesting *if* Dehnadi and Bornat have found a predictive test as they describe.

4. RESEARCH METHOD

In this section, we discuss the methodology used in our study.

4.1 Hypothesis

In this study, we examined the predictive power of a student's mental model for his or her success in learning introductory programming; the hypothesis is that there is a positive correlation between a student's mental model and the student's ability to learn programming. The specific research question we investigated is the following:

> *Is there a correlation between the students' consistency in the mental model applied in questionnaire and their performance in the final exam of a seven-week introductory, model-based, object-oriented programming course?*

4.2 The Course

The programming course spans the first half of CS1 at the University of Aarhus. The course runs for seven weeks; two weeks after the course ends, there is a lab examination with binary pass/fail grading. The grading is based solely upon the final examination; acceptable performance during the course is a prerequisite for the final exam but does not count as part of the grading.

Aims: The purpose of the course is for students learn the foundation for systematic construction of simple programs and, through this, obtain knowledge about the role of conceptual modeling in object-oriented programming. The goal is that students become familiar with a modern programming language, fundamental programming language concepts, and selected class libraries.

Competencies: After the course, students should be able to explain and use fundamental elements in a modern programming language, use conceptual modelling in relation to preparing simple object-oriented programs, implement simple object-oriented models in a programming language, and use selected class libraries.

Form: The course runs for seven weeks; every week, there are four lecture hours and four lab hours with a TA. In addition to the scheduled hours, students are supposed to work approximately seven hours per week in study groups or on their own. There is a weekly mandatory assignment.

Exam: The examination resembles an ordinary lab session. The students are tested in groups of up to 25 at a time. The effective examination time is 30 minutes (occasionally, for various reasons, we allowed a bit more time); a full hour is scheduled for each group to allow for preparing and finalizing (upload, etc.). Each group receives a different assignment consisting of 10 small progressive programming tasks. In principle, the assignments are identical (they are all instances of the same generic assignment).

There are *two checkpoints* in the assignment: one after task three and one after task eight. The students are instructed to call upon an examiner to demonstrate their solutions when they reach either of the checkpoints. For each student, we noted the elapsed time at both checkpoints as well as when (if) they finished the assignment (*first interval*, *second interval*, and *final time*), thus providing a rough measure of the student's efficiency and competence.

A more detailed description of the course can be found in [7]; an evaluation of the examination can be found in [4].

4.3 Subjects

There are approximately 300 students from a variety of study programmes, e.g. computer science, mathematics, geology, nano science, economy, multimedia, etc. Forty percent of the students are majors in computer science; they are the only group of students that continues with the second half of CS1. The rest of the students proceed to other programming courses related to their fields (e.g. multimedia programming, scientific computing, etc.).

The population for this study was 142 students; of the 150 students who volunteered to participate at the beginning of the course, 142 attended the final exam.

The students answered the questionnaire in the first week before the assignment statement was taught.

4.4 Classification of Mental Model and Exam Result

To determine the consistency of the mental model for each of the students, we used the categorization instrument proposed by Dehnadi [13]. From the 12-tuple that describes the mental models applied by a student in the questionnaire, we divided the students into five categories C_i, $0 \leq i < 5$, of decreasing consistency, C_0 being the most consistent category and C_4 the least consistent category. A student is in consistency category C_0 if at least eight mental models in the student's 12-tuple are identical. For the coarse-grained consistent/inconsistent categorization, students in C_0 are considered consistent while students in any of the other categories are considered inconsistent. For further details, see [13].

The binary pass/fail grading of the exam was too coarse-grained to allow for statistical analysis. Therefore, we subdivided the students into four groups, G_i, $0 \leq i < 4$. G_0 represents the students that failed the exam; G_1 represents the students who barely passed the exam (i.e. reached the second checkpoint in the very last minute), G_2 represents the students who produced an average performance (i.e. reached the second checkpoint in due time but did not finish the assignment), and G_3 represents the students who finished the assignment within the time limit with a program that fulfils the complete specification.

5. FINDINGS

In this section, we present the findings from the questionnaire.

5.1 Results

The distribution between consistent and inconsistent broken down to the exam result and prior programming experience is shown in Table 1.

	Consistent	**Inconsistent**
Total	124	18
Pass at the final exam	120	16
Fail at the final exam	4	2
Prior programming experience	85	2
No prior programming experience	39	16

Table 1: *Number of consistent and inconsistent students*

We might consider breaking the data down to other variables, e.g. gender, major, and seniority (study age); however, from previous research, we know that these do not influence students' performance in this course [6]).

5.2 Programming Aptitude

In order to validate Dehnadi and Bornat's findings, we have used a Pearson correlation coefficient test [23] to find if, for students with no prior programming experience, there is a significant correlation between the consistency level and the grading level (according to the *C*- and *G*-categories described in section 4.4).

The P-value is –0.072. Thus, we concluded that there is *no correlation* between consistency of the mental model and performance in our introductory programming course, i.e. we cannot verify Dehnadi and Bornat's findings. Traditionally, a P-value of at least 0.3 (numerically) is required for correlation. (The negative P-value is expected since C_0 corresponds to the highest level of consistency and C_4 to the lowest level of consistency.)

To take a closer look at this contradictory result, we have tested for correlation for a more fine-grained partitioning than the five competence-levels and four grading levels applied above.

We made a more fine-grained partitioning of the mental models by refining the C_i categories: C_i represents the students' whose maximum number of answers of the same mental model equals *i*, thus providing 13 different categories of mental models. Similarly, we have refined the G_i categories to reflect the students' performance according to the second interval (the time elapsed when reaching the second checkpoint), i.e. G_i is the students for whom the second interval is *i* minutes.

The distribution of the data certainly does not indicate a correlation (see Figure 2). A Pearson correlation test confirms this impression with the same result as before (P=–0.075).

Our result is a clear and unequivocal rejection of the research question: there is absolutely no correlation between students' consistency of the mental model applied in the questionnaire and their performance in the final exam of a seven-week introductory, model-based, object-oriented programming course.

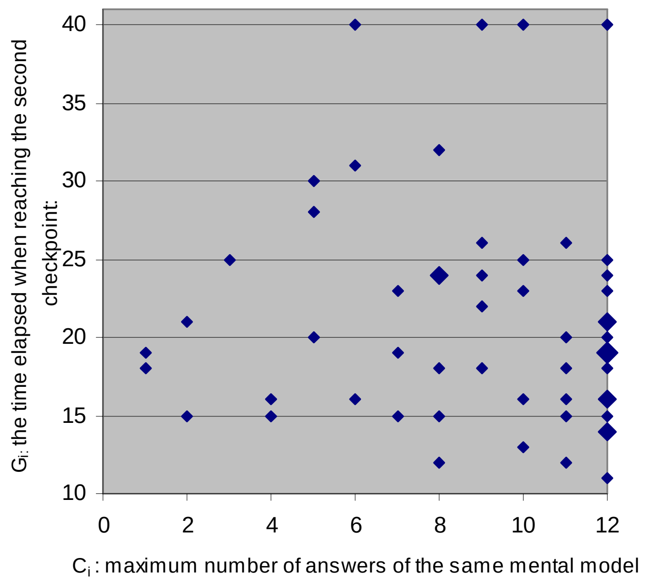


Figure 2: *Second interval versus maximum number of identical mental models in 12-tuple*

If the hypothesis of positive correlation between a student's mental model and ability to learn programming is to be confirmed, it requires an interpretation of the mental model which is different than the one reflected in Dehnadi's questionnaire or another interpretation of ability to learn programming different than the one reflected by the exam of the introductory programming course.

6. DISCUSSION

Our unequivocal result gives rise to a number of questions. One question is whether Dehnadi and Bornat's interpretation of their results is viable. Another question is the validity of the test instrument and speculations about other and better test instruments. But before we address these questions, let us look at possible explanations of our differing results.

6.1 Explaining the Differing Results

The large number of non-fixed variables in Dehnadi and Bornat's investigation and ours allow many explanations. First of all, the investigations have been carried out in different course contexts. We do not know much about the nature of the course at Middlesex University, but it is undoubtly different than ours and may be so in many respects: course material (e.g. textbook, programming language, development environment), course structure (e.g. number of lectures and lab hours), course work (e.g. mandatory assignments, project work), availability of resources (e.g. support material, support for collaboration, student/instructor ratio), and the degree of alignment (concordance between syllabus, course content and the exam). Also, the exam may be different; again, we do not know anything about the nature of the courses at Middlesex University. The instructor is different and may be so in many respects (e.g. teaching experience, familiarity with the subject, personal attitude), and, finally, the students may be different in many ways (e.g. age, study seniority, major).

With all this variation, how can we ever generalise findings from the context where the findings are identified? The best way is by inductive reasoning which can be fuelled by similar findings across a multitude of institutions [16].

6.2 Questioning the Validity of the Test Instrument

Dehnadi and Bornat's interpretation of the students' behaviour in the first test goes as follows: "What distinguish the three groups in the first test is their different attitudes to meaninglessness. The consistent group showed a pre-acceptance of this fact: they are capable of seeing mathematical calculation problems in terms of rules, and can follow those rules wheresoever they may lead. The inconsistent group, on the other hand, looks for meaning where it is not. The blank group knows that it is looking at meaninglessness, and refuses to deal with it."

Contrary to Dehnadi and Bornat, we interviewed our subjects. We conducted individual interviews with the 14 students who were inconsistent but did pass the final exam. They all remembered the test very well. Interestingly, they all started out with some mental model, some set of rules that gave meaning to the "meaningless" notation in the questionnaire. The problem for the 14 students was that the model they started out with failed at some point before the end of the test. Not knowing about the purpose of the test, and not considering it important, none of the students cared to backtrack to find a viable model. They simply altered their model and went on from there. Our harsh conclusion is that it seems as if the only thing the test instrument is testing is the students' guessing capabilities; can they guess a *viable* model up front or can they not? This is hardly an interesting classification of students.

6.3 Alternative Test Instruments

In the light of the conclusion of the previous section, we must reject the test instrument proposed by Dehnadi and Bornat. However, the idea of testing a correlation between the inclination to give meaning to meaninglessness and performance in an introductory programming course, as suggested by Dehnadi and Bornat's comment in their interpretation of their observations, hints at an alternative test instrument.

If the hypothesis is that the inclination to give meaning to meaninglessness is a predictor of success in an introductory programming course, we should devise a test instrument for that. Such a test instrument can easily be constructed by describing a meaningless set of rules and then asking the students to apply these rules to a number of situations. Different test instruments could be constructed: some that invite for interpretation and resulting false applications of the rule set, and some that (by being more neutral) does not. Developing different test instruments along these lines enables tests of the test instruments which in itself is a reasonable task to undertake.

7. CONCLUSION

We tested the hypothesis of a correlation between a student's mental model (according to Dehnadi's definition in [14]) and how well the student performs in an introductory programming course at university. Our result is an unequivocal rejection of the hypothesis.

The result is a surprise—at least in light of [14] in which the authors conclude that the test, although not perfect, is the first test to claim any degree of success.

We have enumerated many explanations for our differing results; in particular, we question the test instrument from [13] used to categorise students according to the mental model, and we suggest a research method from which it may be possible to generalize local results in this area.

A qualitative analysis in the form of interviews with selected subjects has revealed that the test instrument does not seem to measure what it is supposed to; based on insights from the interviews we have devised alternative test instruments.

Our result is encouraging since we do not adhere to the sheep-goat presumption about programming aptitude. To the extent that we shall ever be able to identify concrete factors that predict success, we will use these to improve students' background to increase their chances for success in learning to program. That is our motivation for doing research in this area.

Dehnadi and Bornat's idea of predicting success from mental model is interesting and maybe viable but at least requires an improved test instrument.

8. ACKNOWLEDGMENTS

We thank all the students from the course *Introduction to Programming* at the University of Aarhus in the fall of 2006 who took the time to participate and make this research possible.

9. REFERENCES

[1] C. Arthur. How can I tell if I'll be any good as a programmer? In *The Guardian*, Thursday July 27, 2006.

[2] J. Bennedsen. Teaching Java programming to media students with a liberal arts background. In *Proceedings for the 7th Java & the Internet in the Computing Curriculum Conference* (JICC 7) Monday 27th January 2003, 2003.

[3] J. Bennedsen & M. Caspersen. Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bulletin*, 38(2):39-43, 2006.

[4] J. Bennedsen and M. Caspersen. Assessing Process and Product - A Practical Lab Exam for an Introductory Programming Course. In *Proceedings of the 36th Annual Frontiers in Education Conference*, M4E-16-M4E-21, San Diego, California October 28-31, 2006.

[5] J. Bennedsen and M. Caspersen. An Upcoming Study of Potential Success Factors for an Introductory Model-Driven Programming Course. In *Proceedings for the Fifth Koli Calling Conference on Computer Science Education* pages 166-169, Koli, Finland 18-20 November 2005.

[6] J. Bennedsen and M. E. Caspersen. An investigation of potential success factors for an introductory model-driven programming course. In *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research*, 155-163, Seattle, WA, USA, 2005.

[7] J. Bennedsen and M. E. Caspersen. Programming in context: a model-first approach to CS1. In *SIGCSE '04: Proceedings of the 35th Technical Symposium on Computer Science Education*, 477-481, Norfolk, Virginia, USA, 2004.

[8] S. Bergin and R. Reilly. Programming: factors that influence success. In *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 411-415, St. Louis, Missouri, USA, 2005.

[9] D. F. Butcher & W. A. Muth. Predicting performance in an introductory computer science course. *Communications of the ACM*, 28(3):263-268, 1985.

[10] P. Byrne and G. Lyons. The effect of student attributes on success in programming. In *ITiCSE '01: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, 49-52, Canterbury, United Kingdom, 2001.

[11] C. G. Cegielski & D. J. Hall. What makes a good programmer? *Communications of the ACM*, 49(10):73-75, 2006.

[12] A. T. Chamillard. Using student performance predictions in a computer science curriculum. In *ITICSE '06: Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education*, 260-264, Bologna, Italy, 2006.

[13] S. Dehnadi. Testing programming Aptitude. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, 22-37, Brighton, UK, 2006.

[14] S. Dehnadi and R. Bornat. The camel has two humps. 2006. www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

[15] G. E. Evans & M. G. Simkin. What best predicts computer proficiency? *Communication of the ACM*, 32(11):1322-1327, 1989.

[16] S. Fincher and M. Petre. *Computer Science Education Research*. Routledge Falmer, London, 2004.

[17] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? In *ITiCSE '00: Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, 25-28, Helsinki, Finland, 2000.

[18] R. R. Leeper and J. L. Silver. Predicting success in a first programming course. In *SIGCSE '82: Proceedings of the Thirteenth Technical Symposium on Computer Science Education*, 147-150, Indianapolis, Indiana, United States, 1982.

[19] L. P. McCoy & J. K. Burton. The relationship of computer programming and mathematics in secondary students. *Comput. Sch.* 4(3-4):159-166, 1988.

[20] N. Pillay & V. R. Jugoo. An investigation into student characteristics affecting novice programming performance. *SIGCSE Bulletin*, 37(4):107-110, 2005.

[21] N. Rountree, J. Rountree, A. Robins and R. Hannah. Interacting factors that predict success and failure in a CS1 course. In *ITiCSE-WGR '04: Working Group Reports from Innovation and Technology in Computer Science Education*, 101-104, Leeds, United Kingdom, 2004.

[22] P. Ventura. Identifying predictors of success for an objects-first CS1. *Computer Science Education*, 15(3):223-243, 2005.

[23] L. Wallnau and F. Gravetter. *Essentials of Statistics for the Behavioral Sciences*. Thomson Learning, New York, 2005.

[24] B. C. Wilson and S. Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. In *SIGCSE '01: Proceedings of the thirty-second technical symposium on Computer Science Education*, 184-188, Charlotte, North Carolina, United States, 2001.

15 Exposing the Programming Process

The paper *Exposing the Programming Process* presented in this chapter has been published as a conference paper [Bennedsen et al. 2005a] and as a chapter [Bennedsen et al. 2007b] of the forthcoming book [Bennedsen et al. 2007a].

The book chapter is a revised version of the conference paper. The content of this chapter is equal to the book chapter [Bennedsen et al. 2007a].

[Bennedsen et al. 2005a] Bennedsen, J. and Caspersen, M.E., “Revealing the programming process”, *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, pp. 186-190, 2005.

[Bennedsen et al 2007a] Bennedsen, J., Caspersen, M.E. and Kölling, M., (Eds.) *Reflections on the Teaching of Programming.* Springer-Verlag, 2007.

[Bennedsen et al. 2007b] Bennedsen, J. and Caspersen, M.E., “Exposing the Programming Process”. In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

Exposing the Programming Process[1]

Jens Bennedsen[1] and Michael E. Caspersen[2]

[1] IT University West, Denmark
jbb@it-vest.dk
[2] Department of Computer Science, University of Aarhus, Denmark
mec@daimi.au.dk

Abstract. One of the most important goals of an introductory programming course is that the students learn a systematic approach to the development of computer programs. Revealing the programming process is an important part of this; however, textbooks do not address the issue – probably because the textbook medium is static and therefore ill suited to expose the process of programming. We have found that process recordings in the form of captured narrated programming sessions are a simple, cheap, and efficient way of providing the revelation. We identify seven different elements of the programming process for which process recordings are a valuable communication media in order to enhance the learning process. Student feedback indicates both high learning outcome and superior learning potential compared to traditional classroom teaching.

1 Introduction

We believe that one of the most important goals of an introductory programming course is that the students learn a systematic approach to the development of computer programs.. Revealing the programming process is an important part of this, and we have found that process recordings in the form of screen captured narrated programming sessions is a simple, cheap, and efficient way to provide the revelation. We hereby expand the applied apprenticeship approach as advocated in (Astrachan & Reed, 1995; Linn & Clancy, 1992).

Revealing the programming process to beginning students is important, but traditional *static* teaching materials such as textbooks, lecture notes, blackboards, slide presentations, etc. are insufficient for that purpose. They are useful for the presentation of a product – a finished program– but not for the presentation of the *dynamic* process used to create that product. Besides being insufficient for the presentation of a development process, the use of traditional materials has another

[1] This chapter is based on Bennedsen, J. and Caspersen, M. E. 2005. Revealing the programming process. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (St. Louis, Missouri, USA, February 23 - 27, 2005). p. 186-190

drawback: typically, they are used for the presentation of an *ideal* solution that is the result of a non-linear development process. Like others (Soloway, 1986; J. Spohrer & Soloway, 1986; J. C. Spohrer & Soloway, 1986), we consider this to be problematic; the presentation of the product independently of the development process will inevitably leave the students with the false impression that there *is* a linear and direct "royal road" from problem to solution. This is very far from the truth, but the problem for novices is when they see their teacher present clean and simple solutions, they think they themselves should be able in a straightforward fashion to develop solutions in a similar way. When they realize they cannot, they blame themselves and feel incompetent. Consequently, they will lose self-confidence and in the worst case their motivation for learning to program.

Several tools for visualizing programs exists (e.g. Jeliot (Levy, Ben-Ari, & Uronen, 2003), Alice (Cooper, Dann, & Pausch, 2000)). These tools focus on visualizing the execution of programs and not the development of such programs.

Besides teaching the students about tools and techniques for the development of programs (e.g. a programming language, an integrated development environment (IDE), programming techniques), we must also teach them about the development process (i.e. the task of using these tools and techniques to develop, in a systematic, incremental and typically non-linear way,). An important part of this is to expound and demonstrate that

- many small steps are better than few large ones
- the result of every little step should be tested
- prior decisions may need to be undone and code refactored
- making errors is common also for experienced programmers
- compiler errors can be misleading/erroneous
- online documentation for class libraries provide valuable information, and
- there is a systematic, however non-linear, way of developing a solution for the problem at hand.

We cannot rely on the students to learn all of this by themselves, but using an apprenticeship approach we can show them how to do it; for this purpose we use process recordings.

The chapter is structured as follows: Section 2 is a brief introduction to the notion of process recordings. In section 3 we discuss the need for exposition of the programming process (e.g. through process recordings) and why textbooks are ill suited for this purpose. Section 4 is a more detailed description of process recordings and we identify seven different categories. In section 5 we discuss the use of process recordings in a course context. Section 6 is a brief discussion of related work. The conclusions are drawn in section 7.

2 Process Recordings – a Brief Introduction

Written material in general and textbooks in particular are not a suitable medium through which to convey processes. We have used process recordings, captured and narrated programming sessions, to do that. The creation of a process recording is easy, fast, and cheap, and does not require special equipment besides a standard computer.

The term *process recording* refers to a screen capture of an expert programmer (e.g. the teacher) solving a concrete programming problem, thinking aloud as he moves along. A process recording can be produced using a standard computer; there is no need for a special studio or other expensive equipment. The software for capturing is free, and depending on how advanced post production one needs, that software is either free or very cheap. We have used Windows Media Encoder and Windows Media File Editor, both freeware programs.

We have found that 15-20 minutes is an appropriate duration of a process recording; for some problems the duration can be longer. For convenience, we offer an index (a topic → time mapping) to help retrieve sections of special interest. The index of each recording is stored in a database allowing the students to search for specific material at a later stage. Figure 1 shows a snapshot of a playback of a process recording.

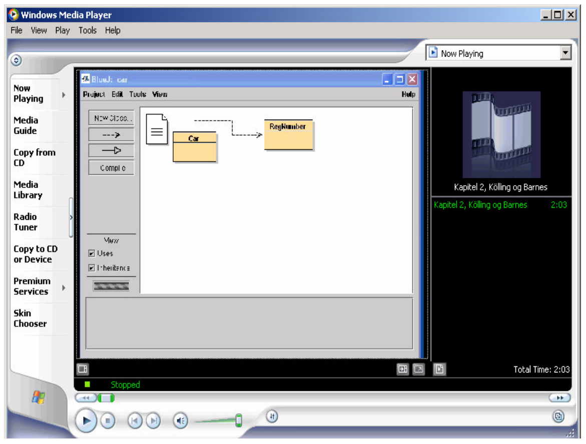


Figure 1: Playback of a process recording

Fig. 1. Playback of process recording

2.1 The Production Process

Most process recordings can be produced without too much preparation. It is our experience that a detailed manuscript is superfluous; too detailed a manuscript tend to make the process recording less authentic and in the worst case plain boring. In one of our distance education introductory programming courses, we have created approximately 60 process recordings. It is our experience that we use one hour to prepare a 30-minute recording and another 20 minutes for post-production.

The technical setup is rather simple. The lecturer sits in front of his computer with a microphone. He starts by introducing the problem and after that talks aloud about the problems he encounter and the possible solutions to these problems. Hereby he makes his programming process explicit – what are the problems, what are the solutions, what are the alternatives …

To increase the motivation for the students we have used some of their solutions as a starting point. One needs to be careful not to belittle the solution, but to show how different techniques can improve an already working solution in order to make it more readable, shorter, more reusable or whatever the focus is.

To increase usability we make it possible for students to navigate in the process recording. The addition of the topic → time mapping has added a new usage of the material: the students can search the material afterwards and use it as yet another part of their learning material repository. In this way, the value of the lectures has expanded from something that is only useful if you are present, to a material that can be used repeatedly over time.

3 Teaching the Process of Programming

The concern for teaching process and problem solving is not new; David Gries (1974)wrote:

Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!

Clearly, cabinet making cannot be taught simply by teaching the tools of the trade and demonstrating finished products; neither can programming! Nevertheless, this seems to be what was being attempted thirty years ago when Gries wrote the above analogy, and largely it seems to be the case today.

du Boulay (1989) identifies *Pragmatics* – the skills of planning, developing, testing, debugging and so on – as an important domain to master. The latter is concerned with skills related to the programming process; however, only few of these are addressed in traditional textbooks on introductory programming.

Caspersen and Kölling(2006) describes in detail how a programming process for novices could be described. The focus on five steps:

1. Create the class (with method stubs)
2. Create tests
3. Alternative representations
4. Instance fields

5. Method implementation
6. Method implementation rules (p. 893-894)

Their focus is on how newcomers can come from a specification (in the form of a UML class) to an implementation fulfilling the specification. In Bennedsen and Caspersen (Model-driven programming, chapter 9 this book) a description of a process for novices for a complete class diagram is discussed.

3.1 Textbooks Neglect the Issue

At a recent workshop (Kölling, 2003) a survey of 39 major selling textbooks on introductory programming was presented. The overall conclusion of the survey was that all books are structured according to the language constructs of the programming language, not by the programming techniques that we (should) teach our students. This is consistent with the findings in (Robins, Rountree, & Rountree, 2003): *Typical introductory programming textbooks devote most of their content to presenting knowledge about a particular language* (p. 141). The prevailing textbook approach will help the students to understand the programming language and the structure of programs, but it does not show the student how to program – it does not reveal the programming process.

We know what is needed, so why has the topic not found its way into textbooks on introductory programming? The best answer is that the static textbook medium is unsuitable for this kind of dynamic descriptions.

3.2 New Technology Allows for Changes

Earlier it has been difficult to present actual programming to students. When programs, in the form of finished solutions, were presented to students it was in the form of writings on the blackboard or copies of finished programs (or program fragments) on transparencies for projection.

Programming on a blackboard has the advantage that it is possible to create programs in dialog with the students at a pace the students can follow; also, the teacher and the students can interact during the development of the program. The obvious drawback is that only small programs can be presented, and neither are we able to run and modify the programs nor to demonstrate professional use of the development tool(s) and programming techniques.

Finished programs on transparencies provide a way of presenting larger and more complex programs to the students, programs that we would never consider writing on a blackboard. This approach has the drawback that teachers tend to progress too fast and exclude the students from taking part in the development.

The emergence of new technology has made it possible in a simple and straightforward manner to present live programming to students. Live programming can be presented in two different ways: live programming using computer and projector, and process recordings showing an expert at work.

Live programming in the lecture theatre using computer and projector is like a combination of using blackboard and slides, but with the important additional ability to run and test the program and to use the programming tools (IDE, online documentation, diagramming tools). This is much closer to the actual programming process than the first two approaches. However, there are still drawbacks: time in the class room is limited and this restricts the complexity of the examples that are presented; also, the presentation vanishes as it takes place; nothing is saved afterwards.

Process recordings showing the programming process of an expert are similar to live programming but without its limitations. In process recordings you can take the time needed to present as complex an example as you wish, and the presentation can be reviewed over and over as many times as a student needs to.

The first three approaches have in common that they are synchronous, one shot events. There is no possibility for the student to go back and review (a step in) the development process if there were something he did not understand. This opportunity is exactly what is added by using process recordings.

4 A Categorization of Process Elements

In this section we present a more detailed description of the process elements we expose through process recordings, and we identify seven different categories that we have found useful in CS1.

A typical programming process encompasses the following process elements:

- Use of an IDE
- Incremental development
- Testing
- Refactoring
- Error handling
- Use of online documentation
- Model-based programming

All are unsuitable for textual descriptions, but important for the student to master. For each process element, we will discuss how to address it in an introductory programming course and how process recordings can be used to reveal its core aspects.

Use of an IDE: We use a simple IDE (Kölling, 1999) . However, a short recording demonstrating the use of special facilities in the IDE makes it still easier for the students to start using it.

Incremental development: Students often try to create a complete solution to a problem before testing it. This is not the behaviour we want the students to exhibit; instead we want them to create the solution in an incremental way taking very small steps alternating between implementing and testing. Following this advice makes it much easier to find and correct errors and it simplifies the whole activity. This topic is very difficult to communicate in a book. With a process recording, it is simple and straightforward to demonstrate how to behave.

Testing: We promote two simple techniques for testing: interactive testing through the IDE (BlueJ) or the creation of a special class with test methods. The process aspect of the former technique is covered under "Use of the IDE" above (see also (Rosenberg & Kölling, 1997)). A textbook is useful for describing principles and techniques for testing but how to integrate testing in the development process is best demonstrated showing a live programming/testing process.

Refactoring: When the students read a textbook they easily get the impression that programmers never make mistakes, that programmers always create perfect, working solutions in take one, and that programmers therefore never have to correct and improve their programs. In (Fowler & Beck, 1999) it is stated that an experienced programmer should expect to use approximately 50% of his time refactoring his code. If this is the case for an experienced programmer, a novice programmer should expect to use significantly more time refactoring/correcting; clearly, students cannot expect to create perfect solutions in take one. But the students get the impression that they ought to be capable of this.

We have found it difficult to motivate the need for refactoring to students. The goal of refactoring is to create better programs in the sense of exhibiting lower coupling and higher cohesion. The students do not know when it is advantageous to refactor a program; they consider the job done when the program can compile and run. But showing them the refactoring techniques "live" gives them a much better understanding of the techniques and an appreciation of the necessity for refactoring. In order to optimise motivation we often start out with a student's program, showing how refactoring can make that program more readable, and how lower coupling and higher cohesion can be obtained through successive applications of simple standard techniques.

Error handling: In order to make the students feel more comfortable it is important to show them that every programmer makes errors and that error handling is a part of the process. It is important to show the students how errors are handled. In particular it is important to demonstrate to the students that the output from the compiler does not always indicate the real error and that there are different types of errors. The process recordings help by being explicit and by dealing systematically with each kind of error.

Online documentation: Modern programming languages are accompanied by large class libraries which the students need to use. The documentation for Java is available online, and the students have to be acquainted with the documentation and how to use it in order to write programs. When the students write code, we force them to write javadoc too. In order to teach how to write and generate the documentation, we show how to do this as an integrated part of the development process using live programming/process recordings.

Model-based programming: We teach a model-driven, objects-first approach as described in Bennedsen and Caspersen, chapter 9. In order to do so the students need to use more than the traditional programming tools; they need to use a tool for describing the class models. The students also need to understand the interaction between the IDE and the modelling tool as well as the relation between model and code. To reinforce the importance of modelling as an integrated part of program development it is vital to show the students the tools.

5 Process Recordings in a Course Context

In this section we will describe how the process recording materials are used in an introductory object-oriented programming course.

5.1 Categories of Process Recordings

We have created five different types of process recordings: introduction to assignments, solutions to the assignments, documentation of synchronous activities (lectures and online meetings), alternative teaching materials, and tool support.

Introduction to assignments: Many students struggle with getting started with an assignment: what is the problem, how shall I start, what exactly is it that I have to do? Many such questions can efficiently be addressed in a process recording where also fragments/structure of a solution can be presented.

Solutions to assignments: Presentation of a solution to a programming assignment; besides presenting the solution, we also present aspects of the development process.

Documentation of synchronous activities: By capturing live programming as it takes place, the students get the opportunity to review (parts of) the process at a later stage.

Alternative teaching materials: For the core topics in the text, we create small programming problems to illustrate the use and applicability of the topic. This provides diversity in the course material supporting different styles of learning.

Tool support: We have created different kinds of process recordings for tool support. Like (Alford, 2003) we have found that, instead of creating written descriptions and manuals for these tasks, it is much easier for us as well as the students if we create a process recording showing how to *do* things: just tell what you are doing on the screen while capturing it.

5.2 Student Feedback

Recently we taught two introductory programming courses based on distance education with respectively 35 and 20 students (a detailed description of the design of this course can be found in (Bennedsen & Caspersen, 2003)). For these courses, we made extensive use of process recordings. All of these materials are stored on a web-server and the students can access them whenever they want and from where they want.

We have evaluated the use of process recordings in our introductory programming course. The evaluation was done quantitatively using a questionnaire as well as qualitatively by interviewing a number of students about their attitude towards the material. From the questionnaire we can see that more than 2/3 of the students have seen more than 50% of the process recordings.

The distribution of hits for the different types of process recordings is as follows: introduction to assignments 28%, solutions to assignments 19%, documentation of synchronous activities 9% alternative teaching materials 21%, and tool support 23%. The interesting thing is that the possibility of reviewing the synchronous activities has by far the smallest hit rate; this indicates that web casting of lectures, which is a widespread use of process recordings (Berkeley, 2007; MIT, 2006), is seen by the students as the least useful of the five categories.

The students have self-evaluated the learning outcome of the process recordings; the result of the evaluation is: No 21%, Small 0%, Ordinary: 21%, High: 14%, Very high: 44%. 58% has indicated a high or very high learning outcome which is very encouraging. In post-course interviews, the students generally confirmed this.

6 Related and Future Work

Streaming video has become more and more popular and common (Ma, Lee, Du, & McCahill, 1996; Smith, Ruocco, & Jansen, 1999). Compression techniques have been standardized and improved; bandwidth is increasing (also in private homes) making it realistic to use videos in an educational setting.

Web casts of lectures is used by many universities including prominent ones like Berkeley and MIT (Berkeley, 2007; MIT, 2006). While such videos may be valuable to students who are not able to attend the lecture or would like to have (parts of) it repeated, they do not significantly add new value to the teaching material.

The use of process recordings in teaching is not new (Smith et al., 1999). Process recordings are used extensively in (D. Gries, Gries, & Hall, 2002), but the use is somewhat different from ours: all process recordings are very short and focused on explaining a single aspect of the programming language or programming; the process recordings are “perfect”, they do not show that it is common to make errors (and how to correct them); and the process recordings do not show the integrated use of the different tools like IDE, online documentation, etc. The process recordings in (D. Gries et al., 2002) can be characterized as alternative teaching materials according to our categorization in the previous section.

Others use a much richer form of multimedia than plain video. One example is the learning objects discussed in (Boyle, 2003). The same differences as described above apply, and on top of that the production cost for creating these learning objects is extremely high.

Much more needs to be done in this area. The overall long-term objective of programming education is that students learn strategies, principles, and techniques to support the process of inventing suitable solution structures for a given programming problem. One possible approach to advance our knowledge is to identify, analyze, and categorize existing methodological and systematic approaches to the practice of programming and programming education — including classical programming methodology of the Dijkstra-Gries school, design by contract, elementary patterns, the existing but scarce literature on the practice of programming, and a study of the practice of masters. Furthermore, it is necessary to identify, categorize, and operationalize strategies, principles, and techniques for object-oriented programming and indicate how these can be made available to novices as well as more experienced programmers through education. Finally, the insight from this work should form the basis of a formulation of requirements for programming environments and languages for programming education.

7 Conclusions

The idea of revealing the programming process is not new:

Anyone with a reasonable intelligence and some grasp of basic logical and mathematical concepts can learn to program; what is required is a way to demystify the programming process and help students to understand it, analyse their work, and most importantly gain the confidence in themselves that will allow them to learn the skills they need to become proficient.

This quotation is fifteen years old (Gantenbein, 1989); nevertheless, the issue still has not found its way into programming textbooks.

Revealing the programming process is an important part of an introductory programming course that is not covered by traditional teaching materials such as textbooks, lecture notes, blackboards, slide presentations, etc. This is just as good since these materials are insufficient and ill suited for the purpose.

We suggest that process recordings in the form of screen captured narrated programming sessions is a simple, cheap, and efficient way of providing a revelation of the programming process. Furthermore, we have identified seven elements included in the programming process. For each of these we have discussed how to address it in an introductory programming course and how process recordings can be used to reveal its core aspects.

From our evaluation of the approach we know that the students use and appreciate the process recordings; some students even find the material superior to traditional face-to-face teaching. The creation of video-mediated materials has proven to be easy and cheap as opposed to other approaches to create learning objects.

The advance of new technology in the form of digital media has made it possible to easily create learning material to reveal process elements that in the past only has been addressed implicitly. The students welcome the new material which has great impact on the students' understanding of the programming process and their performance in practical programming. With new technology, in this case computers and video capturing tools, it becomes possible to store information that represent dynamic behaviour, something which is virtually impossible to describe and represent using traditional tools and materials such as blackboards and books. We are looking forward to further pursue this new opportunity.

References

References

Alford, K. L. (2003). Video faqs - instruction-on-demand. *Procedings of the 33rd Frontiers in Education Conference.* Boulder Colorado. S2e-20-s2e-20.

Astrachan, O., & Reed, D. (1995). AAA and CS 1: The applied apprenticeship approach to CS 1. *SIGCSE '95: Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education,* Nashville, Tennessee, United States. 1-5. from http://doi.acm.org/10.1145/199688.199694

Bennedsen, J., & Caspersen, M. (2003). Rationale for the design of a web-based programming course for adults. *Procedings for the International Conference on Open and Online Learning (ICOOL 2003),* University of Mauritius, Mauritius.

Berkeley. (2007). *UC berkeley webcasts.* Retrieved February 17, 2007, from http://webcast.berkeley.edu/courses/

Boyle, T. (2003). Design principles for authoring dynamic, reusable learning objects. *Australian Journal of Educational Technology, 19*(1), 46-58.

Caspersen, M. E., & Kölling, M. (2006). A novice's process of object-oriented programming. *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications,* Portland, Oregon, USA. 892-900. from http://doi.acm.org/10.1145/1176617.1176741

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *J.Comput.Small Coll., 15*(5), 107-116.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway, & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 57-73). Hillsdale, NJ: Lawrence Erlbaum.

Fowler, M., & Beck, K. (1999). *Refactoring: Improving the design of existing code*. Reading, Massachusetts, USA: Addison-Wesley Professional.

Gantenbein, R. E. (1989). Programming as process: A "novel" approach to teaching programming. *SIGCSE '89: Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education,* Louisville, Kentucky, United States. 22-26.

Gries, D., Gries, P., & Hall, P. (2002). *ProgramLive: Master JAVA programming in a dynamic, self-paced learning environment*Wiley.

Gries, D. (1974). What should we teach in an introductory programming course? *SIGCSE '74: Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education,* 81-89.

Kölling, M. (2003). *The curse of hello world*. Oslo, Norway: Invited lecture at Workshop on Learning andTeaching Object-orientation – Scandinavian Perspectives.

Kölling, M. (1999). Teaching object orientation with the blue environment. *Journal of Object-Oriented Programming, 12*(2), 14-23.

Levy, R. B. B., Ben-Ari, M., & Uronen, P. A. (2003). The jeliot 2000 program animation system. *Computers & Education, 40*(1), 1-15.

Linn, M. C., & Clancy, M. J. (1992). The case for case studies of programming problems. *Communications of the ACM, 35*(3), 121-132.

Ma, W. H., Lee, Y. J., Du, D. H. C., & McCahill, M. P. (1996). Video-based hypermedia for education-on-demand. *Proceedings of the Fourth ACM International Conference on Multimedia,* Boston, Massachusetts, United States. 449-450.

MIT. (2006). *Structure and interpretation of computer programs, video lectures.* Retrieved February 17, 2007, from http://www.swiss.ai.mit.edu/classes/6.001/abelson-sussman-lectures/

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

Rosenberg, J., & Kölling, M. (1997). Testing object-oriented programs: Making it simple. *SIGCSE '97: Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education,* San Jose, California, United States. 77-81. from http://doi.acm.org/10.1145/268084.268115

Smith, T., Ruocco, A., & Jansen, B. (1999). Digital video in education. *SIGCSE '99: The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science*

Education, New Orleans, Louisiana, United States. 122-126. from http://doi.acm.org/10.1145/299649.299715

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

Spohrer, J., & Soloway, E. (1986). Analyzing the high-frequency bugs in novice programs. In E. Soloway, & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 230-251). Washington, DC, USA: Ablex Publishing Corporation.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

16 A Novice’s Process of Object-Oriented Programming

The paper *A Novice’s Process of Object-Oriented Programming* presented in this chapter has been published as a conference paper [Caspersen et al. 2006a].

[Caspersen et al. 2006a] Caspersen, M.E. and Kölling, M., “A novice's process of object-oriented programming”, *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications,* Portland, Oregon, USA, pp. 892-900, 2006.

A Novice's Process of Object-Oriented Programming

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

Michael Kölling
Computing Laboratory
University of Kent
Canterbury, Kent CT2 7NF
United Kingdom
mik@kent.ac.uk

Abstract
Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming which appear to deal with 'program' as a noun rather than as a verb.

We present a set of principles and techniques as well as an informal but systematic process of decomposing a programming problem. Two examples are used to demonstrate the application of process and techniques.

The process is a carefully down-scaled version of a full and rich software engineering process particularly suited for novices learning object-oriented programming. In using it, we hope to achieve two things: to help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the aspects of software engineering.

Categories and Subject Descriptors D1.5 [**Programming Techniques**]: Object-oriented programming.
D2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming, Structured programming, Top-down programming*.

D2.4 [**Software Engineering**]: Software/Program Verification – *Class invariants, Programming by contract*.

General Terms Algorithms, Design, Documentation, Languages.

Keywords CS1, Systematic Programming, Programming Process, Design by Contract, Representation Invariant, Objects-First, Stepwise Refinement, Top-Down Design, Incremental Development, Testing, Refactoring, Programming Education, UML, Pedagogy.

1. Introduction

I remember when I first learned to program. I had a couple of workbooks covering the fundamentals of programming. I went through them pretty quickly. When I had done that, I wanted to tackle a more challenging problem than the little exercises in the book. I decided I would write a Star Trek game…
My process for writing the programs to solve the workbook exercises had been to stare at the problem for a few minutes, type in the code to solve it, then deal with whatever problem arose. So I sat confidently down to write my game. Nothing came! I needed to do something beyond coding. But I didn't know what else to do.
Kent Beck [3]

Most texts used to teach beginners to program focus on presenting language constructs, programming language concepts, and computer programs (complete or partial). They are concerned mostly with 'program' as a noun rather than as a verb. The process of program development is often merely implied rather than explicitly addressed. A typical structure is the presentation of a problem followed by a presentation of a program to solve that problem and a discussion of the program's elements.

From the viewpoint of a student, the program was developed in a single step, starting from a problem specification and resulting in a working solution. Sometimes, a semi-formal requirements specification is included as an interim step, but this does not fundamentally alter our main point: the process of software development is essentially invisible. The fact that we all start by developing sub-optimal and partial implementations on our way to a solution, which we later refine and improve, often seems to be the best kept secret of the computing profession.

The exercises in texts often compound the problem; they frequently require small, easy-to-understand steps that are quite different in character from the development of a complete software solution. The problems resulting from this approach are potentially two-fold:

- Students may be able to understand every separate construct but do not have the skills to put the constructs together in an organised way. This is succinctly illustrated in Kent Beck's quote above. Or, if they do succeed:
- Students, who labour through various incorrect attempts at solving a problem, slowly improving their solution, running into regular bugs along the way before developing a solution that mostly works, often think they are poor programmers for experiencing so much trouble along the way.

To solve these problems we need to do two things:

- Teach students about the *process* of software development, to enable them to follow organised steps to move toward a solution to a problem, and
- Treat software development explicitly as a process that is carried out in *stages and small steps*, rather than the writing of a single, monolithic solution.

Copyright is held by the author/owner(s).
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

If we do not explicitly teach the programming process, we end up with two groups of students: those who cannot cope with the challenge of development and those who discover their own process.

Developing software is, by its very nature, always a process, whether we are formally aware of it or not.

Some of the first group, those students we lose, might have been saved had we given them better techniques to address this problem.

Students in the second group can also greatly benefit from a systematic process, since the techniques they discover and apply in an ad-hoc manner often (and unsurprisingly) lead to inadequate and badly designed solutions. The most applied development technique among students is probably the "first solution that comes to mind" technique. Many of our students are so happy to find any solution at all that it does not occur to them to investigate alternatives. Thus, a systematic process should not only help those students who have fundamental problems arriving at any solution at all but should improve the quality of solutions of all students.

The problem has been first identified a long time ago [9, 13, 22]. The terms *stepwise refinement* and *top-down design* were introduced in the 1970s, and the general principles appeared in some texts at the time, but few current texts do justice to the topic. Recently, some work in this area has been published. For example, Bennedsen & Caspersen argue for the necessity of teaching a systematic programming process and demonstrate ways to apply programming strategies and techniques [5]. A further discussion of related work is provided in section 5.

In this paper, we identify and describe systematic programming techniques particularly suited for novices learning object-oriented programming. More specifically, we present part of an informal, but systematic, process of decomposing a programming problem. The process is designed to be applied by beginners. This paper does not completely describe the whole process, but the largest part of it. Some additional work remains to be done.

The aim of this process is to be applied in an introductory learning and teaching situation. Thus, some of the design goals are that the process has very little bureaucratic overhead, is easy to understand, and is simple to follow.

Our hope is that the result is not only to enable more students to develop programs but also to achieve an improvement in code quality (such as readability, correctness, testing, and extendability) of student solutions.

Section 2 presents the techniques in an abstract form, followed by two examples in sections 3 and 4 that illustrate and discuss the techniques. Section 5 discusses related and future work and section 6 presents our conclusions.

2. A SYSTEMATIC PROCESS FOR NOVICES

In this section, we describe, in a general way, some simple steps that can be followed to implement classes whose intended behaviour is essentially understood.

This section is kept brief and is intended as an initial overview – we will discuss the techniques in more detail using an example in the following section.

Our techniques do not address the analysis phase or the finding of the classes from the problem domain. This may be achieved by using the noun/verb method or other simple methodologies. More likely, in very early student exercises, the teacher or the textbook will provide the class structure.

2.1 Step 1: Create the class (with method stubs)

We assume that the classes and their observable (public) functionality is understood and given, for example in the form of a Java interface or carefully written *javadoc* comments.

The first step towards implementation is to create an implementation class that implements this interface (or, if the interface is not formally given, provides methods with the intended signatures). The method implementations at this stage are stubs (i.e. minimal method bodies).

For methods that do not return values, the method body is empty. For methods with return values, the method body consists of a single return statement. The value returned is a default value (zero for numbers, null for object types, etc.).

Repeat this for every class in the project.

2.2 Step 2: Create tests

Once method stubs have been defined, test cases can be written for every method. This is commonly done using JUnit [16]. Several educational tools support JUnit testing (e.g. BlueJ and Dr. Java [18, 11]), and in environments that support recording of interactive testing, such as BlueJ [17], the existence of stubs enables the test interaction to be recorded.

Initially, most tests will fail. Details about how these tests should be developed are beyond the scope of this paper and have been discussed elsewhere [4, 15].

2.3 Step 3: Alternative representations

The next step aims at deciding on an implementation representation for the objects to be defined. The representation is defined by the instance fields of the class.

For every class, alternative representations must be considered. These can be as many as a student can think of, but must be *at least two*.

We label each of our candidate representations R_1 to R_n.

Next, we create a *Representation Evaluation Matrix* (REM). A REM is a table with one column for each candidate representation, and one row for each method in our class to be implemented (Table 1). Above the table is a short description of each alternative.

R_1: a short description of the first representation alternative here

R_2: a short description of the first representation alternative here

IMPL. EFFORT	R_1	R_2
method1()	Challenging	Trivial
method2()	Trivial	Hard
method3()	Easy	Hard

Table 1: *Implementation effort estimation matrix*

We use this matrix to compare each method that must be implemented for each possible object representation. The comparison criteria may vary – leading to different tables – but is initially always "implementation effort".

Table 1 shows an example of an *Effort REM*. In this table, we compare the estimated effort it takes to implement each method using a particular object representation. As values, we use a small ordered set of effort qualifiers. They are *Trivial*, *Easy*, *Average*, *Challenging*, and *Hard* (the "*TEACH* scale").

In later exercises, different REMs may be used for other criteria that are explicitly mentioned in the task specification. For example, if runtime performance is an explicitly stated goal, a *Performance REM* may be used.

It is crucial not to judge representations on imaginary requirements. Especially, performance consideration should *not* play a role in early exercises, and it should be made clear that performance is entirely irrelevant for judgement of the Effort REM. We recommend focusing on Effort REMs in early exercises.

Initially the instructor can supply the REM, but gradually the students should be responsible for filling in the REM.

Once the Effort REM is complete, we choose the representation that is judged to have the simplest overall implementation.

2.4 Step 4: Instance fields

When we have settled on one particular representation, we can refine our implementation class.

We now define the fields needed to represent the object. (The field definitions need not be complete; further fields may be added later to support method implementations. However, many important fields are derived from the implementation representation.) The field definitions may include their role (in the form of a comment) and possible constraints on their values (also in comment form).

At this stage, we also provide appropriate initialisations for the fields, either in the form of default values or by using client-supplied values. This includes at least partial implementation of the class's constructor.

2.5 Step 5: Method implementation

Step 5 is actually more than a single step: it has the form of a nested loop. The definition is:

```
while there is an unfinished method:
    Pick an unfinished method;
    Implement the method
```

The "Implement the method" step itself contains a loop:

```
while not done:
    improve the method;
    test
```

The order in which a student chooses the methods is essentially arbitrary. Our recommendation for students who are not entirely confident is to choose the method that, according to the Effort REM, is easiest to implement first.

It is easy to see that this completes the implementation. If a student successfully completes this step, the class is finished.

All the magic now lies in the "Implement method" step. This is still a large task, and needs further advice to break it down into smaller steps.

2.6 Method implementation rules

Implementing a method is potentially a large and non-trivial task. We aim to provide a process that breaks this task into smaller steps as well. This time, we cannot give a single recipe, since details of the method may vary widely. Instead, we give a set of rules that can be applied in certain cases.

Some methods, of course, consist of only a few lines of code and may be easy to write. Our rules aim at breaking all methods down into smaller chunks, until they approach the complexity of those easy-to-write methods. This is essentially a small variation of stepwise refinement [22].

At the heart of this technique is the *Mañana Principle*. The Mañana Principle says

> *When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later.*

Thus, the Mañana Principle encourages separation of concerns and the use of many small methods. We discuss an example below.

To get beginners used to the Mañana Principle, there are some more specific forms of this rule, each of which state a more concrete situation in which this principle should be used. They are:

> ***Special Case rule***: If you write code to treat a special case in your algorithm, treat the special case in a separate method.
>
> ***Nested Loop rule***: If you have a nested loop, move the inner loop into a separate method.
>
> ***Code Duplication rule***: If you write the same code segment twice, move the segment into a separate method.
>
> ***Hard Problem rule***: If you need the answer to a problem that you cannot immediately solve, make it a separate method.
>
> ***Heavy Functionality rule***: If a sequence of statements or an expression becomes long or complicated, move some of it into a separate method.

The special methods created as part of these rules are usually private methods, unless they are created in different classes – we discuss this further below.

It is important to remind students that these separate methods do not need to be implemented straight away. The calling method can be written as if the method existed. Following this, a stub for the Mañana method should be created. (If the programming environment had specific tool support for the Mañana principle, this could be automated by the IDE.)

The specific rules are initially easier to apply, because they provide concrete hints to times when they should be applied. They are, however, just instances of the Mañana Principle, and, if applied regularly, develop a coding habit that encourages the understanding and application of the principle in general.

This principle – and the specific rules – may sound abstract or complicated when presented in this theoretical form, but they are quite easy to understand when presented in the context of an example. In the next section, we discuss the development of a class defining objects for dates (time, month and year) to illustrate these techniques in practice.

3. A FIRST EXAMPLE: DATE

We demonstrate the techniques discussed above in the context of a simple programming problem: the implementation of a class representing a date.

3.1 Specification of Date

Here, we give the specification of the problem as a Java interface. It could easily be presented more informally; the introduction of interfaces is not a requirement for this process.

```
interface Date {
  /**
   * Advance the date to the next day
   */
  void setToNextDate();

  /**
   * Return a string representation of this date
   * in the format yyyy-mm-dd
   */
  String toString();
}
```

Figure 1: *Specification of Date*

3.2 Creating method stubs

The first step is to create a class for the implementation that contains method stubs. The resulting class is presented in Figure 2. (Note that we do not formally implement the interface given above to demonstrate that the use of Java interfaces is not a requirement.)

If the specification was provided in the form of a Java interface, this process is essentially mechanical and could be automated by a development environment. For students in early stages of learning, however, it might help to write this class skeleton by hand. The important thing is: simple rules can be given to guide the creation of this class.

```
/** An instance contains a date */
class Date1 {

  /**
   * Advance the date to the next day
   */
  public void setToNextDate() {
  }

  /**
   * Return a string representation of this date
   * in the format yyyy-mm-dd
   */
  public String toString() {
    return null;
  }
}
```

Figure 2: *Date class with method stubs*

3.3 Test cases

The next step is to ensure that appropriate test cases exist.

Our techniques do not necessarily prescribe a strict test-first approach, in which students create tests for all methods themselves. A viable alternative for early programming tasks is to use teacher-provided tests. The teacher may provide a test suite for the expected methods as part of the specification of the task.

The important step here is to ensure that tests exist, can be compiled, and can be executed (but do not need to pass).

In this paper, we do not present the specific tests, since the actual test development is not the main focus of this paper. The example (including the test), however, is available from a web site. The URL is given at the end of this paper.

3.4 Alternative representations for Date

The next step in our technique is to consider alternative representations (at least two).

An obvious representation for this problem is to use three integer variables *day*, *month* and *year*; we will denote this alternative R_1. An alternative representation is to count the number of days from a certain start date, say 0001-01-01; we denote this alternative R_2.

R_1 simplifies the implementation of *toString* whereas the implementation of *setToNextDate* will be more challenging, since it must deal with the special case of the last day of a month.

R_2 leads to a simple implementation of *setToNextDate* (a simple increment), whereas implementing *toString* will be hard.

The result of this analysis is the Effort REM for Date (Table 2).

***R_1:** Use three integers for date: day: int; month: int; year: int*

***R_2:** Use one integer: number of days since 1 Jan 0001*

IMPL. EFFORT	R_1	R_2
***setToNextDate*()**	Challenging	Trivial
***toString*()**	Trivial	Hard

Table 2: *Estimate of required effort to implement Date*

We choose to use R_1 for our class, since it seems to be the representation that allows for the quickest implementation of *Date*.

3.5 Instance fields of Date

Choosing R_1 as the basis for our implementation determines the instance fields. The definition of class *Date1* after adding the fields is presented in Figure 3. The method stubs are unchanged. Comments from previous code segments are left out for brevity; only comments for new methods are included from here on.

```
class Date1 {

  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  /**
   * Create a date instance with an arbitrary
   * (fixed) value.
   */
  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
  }

  public String toString() {
    return null;
  }
}
```

Figure 3: *Adding instance fields to Date*

3.6 Implementing the methods

The next step is to implement and test the methods. Some methods may be easy to implement in one step; *toString* in our example falls into this category. Other methods may require more work. In this case, partial solutions may be used for initial versions. Figure 4 shows our class after implementing function *toString* and a first, naïve version of *setToNextDate*.

```
class Date1 {
  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
  }
  public String toString() {
    return year + "-" + month + "-" + day;
  }
}
```

Figure 4: *Naïve implementation of Date*

This partial solution is indeed a very naïve implementation. Nevertheless, we might claim that the *setToNextDate* method is 97% correct since it works correctly in 353 out of 365 cases! In some sense, we are very close to a full solution, and if the class is part of a larger system, it can now be used (as a test stub) by other parts of the system.

Incrementing the field *day* might violate the representation invariant, and in this special case the above implementation of *setToNextDate* fails to work properly. We have to check for this special case and handle it appropriately. For simplicity, we temporarily assume 30 days in every month.

In the special case where *day* after being incremented exceeds the number of days in the month, we must set *day* to 1 and increment field *month*. Following our *Special Case* rule from section 2, we deal with this special case by introducing a new private method, *checkDayOverflow*. Figure 5 shows the resulting code.

```
class Date1 {
  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
    checkDayOverflow();
  }

  /**
   * Check for special case where day > daysInMonth;
   * in that case, set day to 1 and add 1 to the month
   */
  private void checkDayOverflow() {
    if ( day > 30 ) {
      day = 1;
      month = month + 1;
    }
  }

  public String toString() {
    return year + "-" + month + "-" + day;
  }
}
```

Figure 5: *Partial implementation of Date*

Now, incrementing the variable *month* might also violate the representation invariant; this special case is handled similarly by introducing a new private method *checkMonthOverflow*, which is called after incrementing *month*. Except for the assumption of 30 days in every month, the method is now finished.

To finish our implementation, we have to replace the literal 30 with the correct number of days in every month. Here, the *Mañana Principle* comes in again, this time in the form of the *Hard Problem* rule: If we need some information that we do not have, we pretend we have a method that gives us the answer. Thus, we just assume a method *daysInMonth* that does exactly what we need. We do not worry about the implementation of this method now; it is postponed until later.

The new version of the *checkDayOverflow* method is shown in Figure 6.

```
private void checkDayOverflow() {
  if ( day > daysInMonth() ) {
    day = 1;
    month = month + 1;
    checkMonthOverflow();
  }
}
```

Figure 6: *Final version of checkDayOverflow()*

This method will not compile until we provide a method stub for *daysInMonth*. The stub, in this case, should not return a zero, but should return 30 – the approximation we have used previously.

The most important thing at this stage is that we have explicitly separated two independent problems: the correct use of this method and the implementation of the method. Separating these problems makes each half easier to solve.

Since our *checkDayOverflow* method is now complete, we might now proceed to implement *checkMonthOverflow*. In the general case, implementing one method may generate several other methods via the Mañana Principle, which can then be gradually implemented.

For our example, implementing the *daysInMonth* method is the last thing that is missing. To calculate the number of days in the current month, we declare a local array variable in this method to hold the number of days per month (with 28 days for February), and the method returns the number of days in the current month by looking up the number in the array. This brings us almost to the finishing line: the implementation now works, except for the special case where the current year is a leap year ("99.93% correctness").

As previously, we treat a special case by introducing a new private method to deal with it. In this case, we introduce a boolean method *isLeapYear* that returns true if the current year is a leap year. The implementation of this method is a straightforward implementation of the leap year rule: a year is a leap year if the year is divisible by 4 but not by 100 or if it is divisible by 400.

The hardest part of this calculation is the check whether a number can be divided by another so, again, following the Mañana Principle, we use a method *divides* that gives us the result, and then we implement that method later.

The complete implementation of our Date class including these methods is shown in Figure 7.

```
class Date1 {
  private int day;   // 1 ≤ day ≤ daysInMonth
  private int month; // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
    checkDayOverflow();
  }
  private void checkDayOverflow() {
    if ( day > daysInMonth() ) {
      day= 1;
      month= month + 1;
      checkMonthOverflow();
    }
  }

  /**
   * Check for special case where month > 12;
   * in that case, set month to 1 and add 1 to the year
   */
  private checkMonthOverflow() {
    if ( month > 12 ) {
      month= 1;
      year= year + 1;
    }
  }

  /**
   * Return the number of days in the current month
   */
  private int daysInMonth() {
    // month:            1  2  3 ... 12
    int[] daysInMonth = {31,28,31,...,31};

    int result = daysInMonth[month-1];
    // special case: February in a leap year
    if ( month == 2 && isLeapYear() ) {
        result= result + 1;
    }
    return result;
  }

  /**
   * Return true iff the current year is a leap year
   */
  private boolean isLeapYear() {
    return (divides(4, year) && !(divides(100, year))
           || divides(400, year);
  }

  /**
   * Return true iff a divides b
   */
  private boolean divides(int a, int b) {
    return b % a == 0;
  }

  public String toString() {
    return year + "-" + month + "-" + day;
  }
}
```

Figure 7: *Complete implementation of Date*

3.7 Discussion of Date implementation

The above development of a class implementing *Date* demonstrates the application of the techniques set out in section 2. The most relevant observation is that every step is broken into small, manageable chunks.

Some of the steps in our technique are fairly easy to learn (creating method stubs, defining the instance fields after deciding on a representation); others require much practice (creating tests, implementing methods).

The detailed discussion of the method implementation has shown that – at least in this case – the harder tasks can also be broken down into small parts. This technique can be applied to any implementation of a method.

4. A SECOND EXAMPLE: CALENDAR

Our second example is one that consists of two classes: a class *Appointment* to record personal appointments, and a class *Calendar* class to hold the appointments. We discuss this example to illustrate some additional points (while mostly skipping those parts that we have already covered above).

4.1 Specification of Calendar

Again, we give the specifications of *Calendar* and *Appointment* in the form of Java interfaces (Figure 8). Alternatively, they may be provided as a UML diagram or informally as a list of required methods.

To abstract from the details of actual time and date information, start time and duration of appointments are represented as integer values. For example, we may have an appointment that starts at 6 and has duration 5 and another that starts at 9 and has duration 1.

```
/** A calendar that can hold appointments */
interface Calendar {

  /** Add appointment a to this calendar */
  void add (Appointment a);

  /** Remove appointment a from this calendar */
  void remove(Appointment a);

  /** Return the first free slot of
      duration d at or after time s */
  Appointment getFirstAvailable(int s, int d);
}

/** An appointment */
interface Appointment {
  int getStartTime();
  int getDuration();
  String getDescription();

  /** collidesWith is true iff this and a overlap */
  boolean collidesWith(Appointment a);
}
```

Figure 8: *Specification of Calendar and Appointment*

4.2 Creating method stubs and test cases

For this example, we skip the discussion of method stub creation and test case definitions, since the process is essentially the same as in the first example. Instead, we jump straight ahead to the discussion of representation alternatives.

4.3 Alternative representations for Calendar

As always, before embarking on implementing a specification, alternative representations must be considered. This must be done for each class. In this discussion, we consider only the implementation of class *Calendar* and ignore class *Appointment*.

One representation of a calendar is an unordered set of appointments; we will denote this representation R_1. An alternative representation is a sorted set of appointments; we will denote this representation R_2.

For both R_1 and R_2, implementation of *add* and *remove* is trivial (a delegation to the similar *Set* method).

R_1 simplifies the programming task of *getFirstAvailable* (at the expense of runtime efficiency). The method can be implemented as a simple linear search where each repetition requires another repetition over the set of appointments (i.e. *getFirstAvailable* will be $O(n^2)$ where n denotes the number of appointments in the calendar), but the required programming effort is manageable.

We know that R_2 allows for a more efficient implementation (*getFirstAvailable* will have time complexity $O(log(n) + m)$ where m denotes the number of collisions until a free slot is found), but clearly this is at the expense of a considerable increase in the complexity of the programming task. R_2 requires the definition of a total ordering (natural order) of appointments as well as fluency with the *SortedSet* interface, which is an order of magnitude more complex than the more straightforward *Set* interface.

The result of the analysis is summarized in the Effort REM for Calendar (Table 3).

R*₁*: Use unordered set to store appointments

R*₂*: Use a sorted set to store appointments

IMPL. EFFORT	R_1	R_2
***add*()**	Trivial	Trivial
***remove*()**	Trivial	Trivial
***getFirstAvailable*()**	Average	Challenging

Table 3: *Estimate of required effort to implement Calendar*

We choose R_1 because it clearly allows for the simplest implementation of *Calendar*.

4.4 Implementation of Calendar

Having decided upon a representation of a calendar (i.e. having defined the representation invariant), we have decoupled the three subtasks of implementing the methods of the *Calendar* interface. This is an instance of the principle *separation of concerns* – Dijkstra's mantra and primary instrument of thought [10, pp. 209-217].

Having decided upon a set representation, where we are free to choose any concrete class that implements the *Set* interface, we can make a partial implementation of *Calendar* (Figure 9).

```
/** A calendar with appointments */
class CalendarUnsorted {
  private Set<Appointment> appointments;

  /** Create an empty calendar */
  public CalendarUnsorted() {
    appointments = new HashSet<Appointment>();
  }

  public void add(Appointment a) {
    // FixMe
  }

  public void remove(Appointment a) {
    // FixMe
  }

  public Appointment getFirstAvailable(int s, int d) {
    return null;  // FixMe
  }
}
```

Figure 9: *Partial implementation of Calendar*

This is indeed a very small step toward a complete implementation of *Calendar*, but it compiles and maybe even makes a few test cases run. For novices (and indeed for others), making small successful steps toward the goal is a rewarding and satisfying way of developing software.

Using a set as the representation of a calendar allows for a straightforward implementation of each of the three methods independently of each other.

Methods *add* and *remove* can be implemented simply by delegating the method call to the similar *Set* methods. Adding this to the initial implementation gives the next two methods of our solution to the problem (Figure 10).

```
/** Add appointment a to this calendar */
public void add(Appointment a) {
  appointments.add(a);
}

/** Remove appointment a from this calendar */
public void remove(Appointment a) {
  appointments.remove(a);
}
```

Figure 10: *Implementation of methods add and remove*

Method *getFirstAvailable* is somewhat more complicated. It can be implemented as a linear search by successively checking for availability of appointment slots (s, d), (s+1, d), (s+2, d), ... until an available appointment slot is found ((s, d) denotes the appointment with start time s and duration d). A first attempt at implementing *getFirstAvailable* is shown in Figure 11.

```
/** Return the first free slot of
   duration d at or after time s */
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  boolean available = false;
  do {
    result = new Appointment(s++, d);
    // set 'available' such that available holds iff
    // result does not collide with any appointment
    // already in the calendar
  } while ( !available );
  return result;
}
```

Figure 11: *Partial implementation of method getFirstAvailable*

It is obvious that the calculation of *available* involves an iteration over the appointments in the calendar, and consequently a nested loop. One of our rules for method implementation is the *Nested Loop* rule: *use a new private method to unfold nested loops*. Instead of proceeding with development of the inner loop, we define a new private method for the calculation of the boolean expression *available* as defined above. We name the method *isAvailable* (Figure 12).

```
/** Return true iff Appointment a does not collide
   with any appointments in this calendar */
private boolean isAvailable(Appointment a) {
  return true; // FixMe
}
```

Figure 12: *Specification of method 'isAvailable'*

With method *isAvailable* to serve us, we can now finish the loop body of method *getFirstAvailable* (Figure 13).

```
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  boolean available = false;
  do {
    result = new Appointment(s++, d);
    available = isAvailable(result);
  } while ( !available );
  return result;
}
```

Figure 13: *Implementation of method getFirstAvailable*

Removing the unnecessary variable *available* gives the final version of *getFirstAvailable* (Figure 14).

```
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  do {
    result = new Appointment(s++, d);
  } while ( !isAvailable(result) );
  return result;
}
```

Figure 14: *Improvement of method getFirstAvailable*

(Side note: we assume here an unbounded calendar, i.e. there will always be an available slot, and the loop will always terminate.

For a bounded calendar, we would have to add a test for reaching the end of the calendar in the loop condition. This would, of course, again involve the Mañana Principle, and we would use a method a*tCalendarEnd*.)

Now we only need to implement the new private method *isAvailable*. As mentioned earlier, this can be done by a repetition checking for collision between *a* and each appointment *i* in the set (Figure 15).

```
private boolean isAvailable(Appointment a) {
  for ( Appointment i : appointments ) {
    if ( a.collidesWith(i) ) return false;
  }
  return true;
}
```

Figure 15: *Implementation of method 'isAvailable'*

This completes the development of an implementation of *Calendar* based on R_1. The development of an implementation of *Appointment* is left to the reader.

4.5 Discussion of development of Calendar

The discussion of the calendar example has shown the application of the *Nested Loop* rule. When consistently applying this rule, the code remains considerably simpler (and easier to understand for beginners) than an alternative using a nested loop.

In this example, all the methods introduced through our rules were private methods in class *Calendar*. In the general case, this does not always have to be the case. If, for instance, class *Appointment* did not have a method *collidesWith*, this method may have been introduced by applying the *Hard Problem* rule while implementing the calendar's *isAvailable* method.

In early exercises, we usually start with problems where the methods that naturally develop are in the same class. This can then – a bit later – be extended and linked to a discussion of responsibility-driven design, and the question which class should provide a new, required method.

5. RELATED AND FUTURE WORK

Numerous software engineering topics relate to our efforts of identifying a systematic programming process for novices. We will discuss these topics in turn.

Stepwise refinement. More than 35 years ago Dijkstra and Wirth identified the need for a constructive and systematic approach to programming – not only for novices but for the community as a whole [8, 9, 22, 23]. Our work builds on the work of Wirth and Dijkstra but concentrates on a specialized process for novices learning object-oriented programming.

Programming methodology. In the early seventies Dijkstra formalized his ideas about structured programming and developed a methodology for systematic construction of programs using functional specifications (pre and post conditions) and loop invariants to drive the development process [10]. In continuation of Dijkstra's seminal work, Back developed a refinement calculus [1, 2] while Gries and others produced text books based on the methodology (e.g. [6, 14, 20]). Our approach differs from this work by being a formally-based but informally-practiced approach to systematic program development.

Responsibility-driven design. The Mañana Principle is related to responsibility-driven design [21]. In this paper, we apply the Mañana Principle only for functional decomposition, but even here it reveals its relationship to responsibility-driven design (the nested loop rule factors a part of the program to a separate method with the responsibility of implementing the nested loop functionality).

Refactoring. During a programming session, it is inevitable that decisions made earlier in the session need to be altered at a later stage. Realizing and learning that this is the rule rather than the exception helps novice programmers come to terms with the fact that programming is not a linear process. This is refactoring-in-the-small [12]. An interesting aspect here is programming environment support: in a similar manner in which refactoring is now commonly supported in development environments, the Mañana Principle could easily be supported by automating the creation of method stubs whenever a new private method is introduced.

XP and agile software development. Extreme programming and agile software development covers many aspects of software engineering [3, 19]; two of the basic principles are: "*Take small steps*" and "*Always do the simplest thing that will work*". We use these principles as guidelines for choosing among several possible implementations of an abstraction (a method specification or an interface) and for the process of implementing it. They are wise guidelines for novices as well as experts.

Test-driven development. The strategy of test-driven development [4, 15] relates closely to step 2 in our process: Create tests. Test-driven development is gaining increased recognition, and it is beneficial to apply this strategy with novices for several reasons (e.g. force a consumer view as well as producer view of program components). But it is not necessary to adopt test-driven development in order to apply our process; instead test cases can be provided as part of the specification of a programming task.

In this paper, we have concentrated on a part of the process where decomposition generates support methods. This part is not exclusively object-oriented and is equally applicable to functional and procedural languages, even though we have presented it in the context of an object-oriented language. Future work includes extending the set of rules that unfolds the Mañana Principle to cover cases of decomposition that generate not only new methods but also new classes (or interfaces).

A second direction of future work will focus on investigating and designing tool support for the process in general and in particular for the Mañana Principle.

6. CONCLUSIONS

We have argued that we need to teach novices about the process of software development in order to enable them to follow organised steps to move toward a solution to a problem, and that we must treat software development explicitly as a process that is carried out in stages and small steps, rather than the writing of a single, monolithic solution.

Furthermore we have identified and described principles and systematic programming techniques particularly suited for novices learning object-oriented programming. To complement the principles and techniques, we have presented an informal but systematic process designed to be applied by beginners. Through two examples we have demonstrated the application of the process.

The process we propose is a carefully down-scaled version of a full and rich software engineering process. By using it we hope to achieve two things: To help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the various aspects of a software engineering process.

The complete programs discussed in this paper are available at www.daimi.au.dk/~mec/oopsla2006/.

7. Acknowledgement

It is a pleasure to thank David Gries for numerous careful comments and improvements to an earlier version of the paper.

References

[1] Back, R.-J., *On the Correctness of Refinement Steps in Program Development*, PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[2] Back, R.-J., *Refinement Calculus: A Systematic Introduction*, Springer-Verlag, 1998.

[3] Beck, K. *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[4] Beck, K., *Test-Driven Development by Example*, Addison-Wesley, 2003.

[5] Bennedsen, J. and Caspersen, M.E., ”Revealing the Programming Process”, *Proceedings of the thirty-sixth SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 2005, pp. 186-190.

[6] Cohen, E., *Programming in the 1990's*, Springer-Verlag, 1990.

[7] Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A:R., *Structured Programming*, Academic Press, 1972.

[8] Dijkstra, E.W., “A Constructive Approach to the Problem of Program Correctness”, BIT 8, 1968.

[9] Dijkstra, E.W., “Notes on Structured Programming”, EWD 249, 1969. In [7].

[10] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[11] Dr. Java, http://drjava.org, Accessed 12 July 2006.

[12] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[13] Gries, D., “What Should We Teach in an Introductory Programming Course”, *Proceedings of the fourth SIGCSE Technical Symposium on Computer Science Education*, 1974, pp. 81-89.

[14] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[15] Hunt, A. and Thomas, D., *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2003.

[16] JUnit. www.junit.org.

[17] Kölling, M., *Unit Testing in BlueJ*. www.bluej.org/tutorial/testing-tutorial.pdf. Accessed 12 July 2006.

[18] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., “The BlueJ System and its Pedagogy”, Computer Science Education, Vol. 13, No. 4, 2003, pp. 249-268.

[19] Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices*, Prentice-Hall, 2003.

[20] Morgan, C., *Programming from Specifications*, Prentice-Hall, 1990. http://users.comlab.ox.ac.uk/carroll.morgan/PfS/ Accessed 12 July 2006.

[21] Wirfs-Brock, R. and McKean, A., *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.

[22] Wirth, N., “Program Development by Stepwise Refinement”, *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227.

[23] Wirth, N., *Systematic Programming*, Prentice-Hall, 1973.

17 CS1: Getting Started

The paper *CS1: Getting Started* presented in this chapter has been published as a conference paper [Caspersen et al. 2000] and as a chapter [Caspersen et al. 2007b] of the forthcoming book [Bennedsen et al. 2007a].

The book chapter is a revised version of the conference paper. The content of this chapter is equal to the book chapter [Bennedsen et al. 2007a].

[Bennedsen et al. 2007a] Bennedsen, J., Caspersen, M.E. and Kölling, M., (Eds.) *Reflections on the Teaching of Programming.* Springer-Verlag, 2007.

[Caspersen et al. 2000] Caspersen, M.E. and Christensen, H.B., "Here, there and everywhere — on the recurring use of turtle graphics in CS1", *ACSE '00: Proceedings of the Australasian Conference on Computing Education,* Melbourne, Australia, 2000, pp. 34-40.

[Caspersen et al. 2007b] Caspersen, M.E. and Christensen, H.B., "CS1: Getting Started". In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

CS1: Getting Started

Michael E. Caspersen and Henrik Bærbak Christensen

Department of Computer Science, University of Aarhus, Denmark
{mec, hbc}@daimi.au.dk

Abstract. The Logo programming language implements a virtual drawing machine—the turtle machine. The turtle machine is well-known for giving students an intuitive understanding of fundamental procedural programming principles. In this paper we present our experiences with resurrecting the Logo turtle in a new object-oriented way and using it in an introductory object-oriented programming course. While, at the outset, we wanted to achieve the same qualities as the original turtle (understanding of state, control flow, instructions) we realized that the concept of turtles is well suited for teaching a whole range of fundamental principles. We have successfully used turtles to give students an intuitive understanding of central object-oriented concepts and principles such as object, class, message passing, behaviour, object identification, subclasses and inheritance; an intuitive understanding of recursion; and to show students the use of abstraction in practice as the turtles at a late stage in the course becomes a handy graphics library used in a context otherwise unrelated to the turtles.

1 Introduction

It is our firm conviction that the primary aim for an introductory programming course is that students learn fundamental programming principles and techniques. The mastery of a programming language is, of course, necessary, but we view it as a secondary concern; we want to focus on fundamental principles and general techniques as early as possible and thereafter unfold these throughout the course.

Contrary to this, most introductory programming texts focus on the programming language, often described in a bottom-up fashion starting with the simpler constructs of the language and progressing to more advanced constructs. Only subordinate to the presentation of the language constructs follows the presentation of programming techniques; however, all too often these programming techniques are not even explicit in textbooks.

Another motivation for our approach is that most people learn more easily through the concrete towards the abstract [5,9]. Having seen constructs and techniques being applied in an appealing intuitive way, and thereafter mimicking these to solve similar problems, like in a craft's apprenticeship, provides an excellent basis for a later thorough and more abstract treatment. In this way the students have a practical experience to ground the abstract treatment.

1.1 The Inverted Curriculum

Our view is not a novel one as is evident from many papers from past SIGCSE conferences [4, 6, 7, 11, 12]. Bertrand Meyer [8] coined the term "the inverted curriculum" (or "consumer-to-producer-strategy") meaning that important topics and concepts should be covered first by using classes solely through their abstract specifications, and only then the students learn about the internals of classes. A simplified variant of Meyer's vision is the objects-first approach which is prevailing in many new textbooks, but still many of these books are structured on the basis of the constructs in the programming language and not on the basis of the language independent concepts, principles and techniques that the students are supposed to master by the end of the course.

Of course, in order to be able to focus on programming techniques and apply these in concrete programs, it is necessary to be—at least to some extent—fluent in a programming language. However, we do not want the learning of the language to take over and become the primary concern, especially not in the beginning of the course. What we want is to jump start the students so that they, as early as possible, can start writing interesting and challenging programs based on the fundamental principles and techniques that are our primary concern in the course: programs as physical models, objects, behaviour, classes, state, control flow, parameterisation, design by contract (specifications), inheritance, etc.

In order to facilitate a jump start of CS1,w e have developed a Java package, Turtles, that takes as it starting point the familiar turtle graphics developed by Seymour Papert and others at MIT in 1967 [10,1]. We use it to give an intuitive introduction to concepts such as state, control flow, and parameterisation. Somewhat to our surprise, it turned out that the Turtles package could play many more roles within CS1 than initially anticipated: It has become a recurring vehicle for introducing such diverse topics as objects and classes, object models, recursion, polymorphism and class hierarchies. Indeed, turtles popped up here, there, and everywhere...

In the current version of our introductory programming course we are using the programming language Java which is also the language of choice for the presentation in this paper.

2 The Turtle Machine

The original Logo turtle machine is a virtual drawing machine that uses the metaphor of a turtle with a coloured pen moving around in a Cartesian drawing area to produce drawings. The state of the turtle machine can be described as a 4-tuple: a turtle position (x, y)-coordinates, an angle, a colour and an up/down status for the pen. Initially the turtle is placed in the lower left corner (0, 0) , the angle is zero, the colour is black and the pen is down (figure 1).

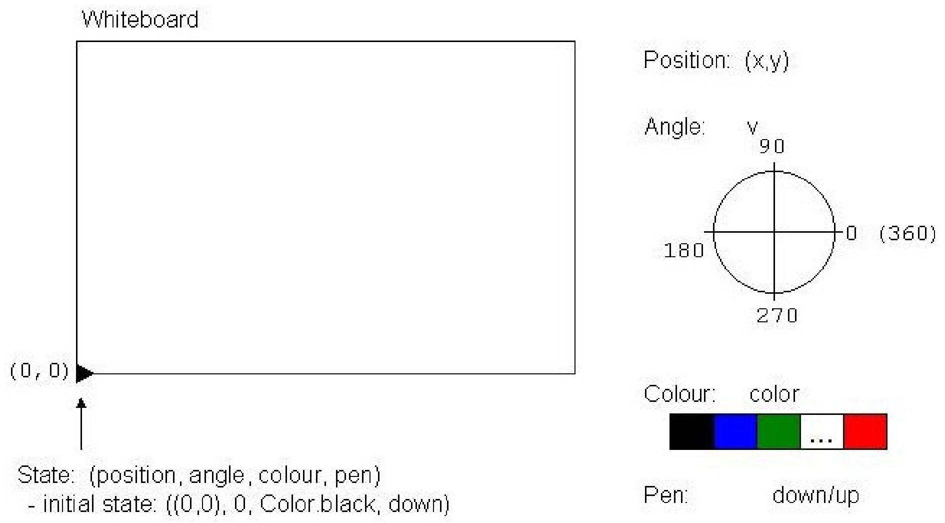

Whiteboard
(0, 0)
State: (position, angle, colour, pen)
- initial state: ((0,0), 0, Color.black, down)
Position: (x,y)
Angle: v
90
180
0 (360)
270
Colour: color
...
Pen: down/up

Fig. 1. Architecture of the Turtle Machine.

The set of instructions for the machine is minimal; only nine instructions are used to operate the machine (see Table 1).

Table 1. Instruction set for the Turtle Machine

Command	**Behaviour**
`move(l)`	move l units in current direction
`moveTo(x, y)`	move to position (x, y)
`turn(d)`	increase the angle d degrees
`turnTo(d)`	set the angle to d degrees
`center()`	move to center
`penUp()`	lift the pen
`penDown()`	lower the pen
`setColour(c)`	set the pen's colour to c
`clear()`	clean the drawing area

3 The Turtle Machine Resurrected: Turtles

The original turtle machine sprang out of the procedural programming paradigm that views a program as a sequence of instructions carried out by some virtual machine. In contrast the object-oriented programming paradigm views a program as a model where model elements are objects that have behaviour and interact with other objects. Thus—in our object-oriented CS1 course—the turtle machine has naturally been replaced by turtle objects. In our Java implementation, there is no machine that executes turtle commands; instead there are objects that exhibit turtle behaviour; behaviour that is described by the Turtle class. The instruction set in Table 1 is replaced by (otherwise semantically equivalent) methods in the Turtle class.

This change of view and paradigm comes natural because the original metaphor of a turtle moving around on a drawing area is inherently an object-oriented model.

4 Jump Starting

At the beginning of the course we teach the concepts from the concrete towards the abstract. We start by introducing our "mascot" turtle with the odd, but short, name `t`. `t` lives in a sandbox (the large drawing area) and has a pen that leaves a trail when it moves around. `t` has *behaviour*: move-,turn-, and pen-behaviour. `t` exhibits the move-,turn-, and pen-behaviour when we pass it the message to do so, e.g. `t.move(100)` tells `t` to move 100 units forward. Before we show a computerised turtle, we actually let the audience command the lecturer around the floor in an attempt to produce a rectangle—while it reinforces the intuitive understanding of the behaviour concept, it also 'breaks the ice' between audience and lecturer as the audience for a short period is 'in control of the lecturer' as they pass messages: "Henrik, please move 2 meters" and so on. Controlling the turtle (or lecturer) also brings an intuitive understanding of the importance of the sequencing of messages passed, the control flow. Parameterisation also follows naturally as e.g. the 'move'-behaviour needs additional detail, namely the actual distance to travel.

The computerised turtle is then described through online viewing, editing , and running of Java code using a laptop computer connected to a projector.

We motivate loops in control flow in order to avoid textual repetition, e.g. looping four times over `{t.move(100); t.turn(90);}` is easier than writing eight turtle messages. This quickly leads to quite interesting drawings as illustrated in figure 2 that is produced by program 1.

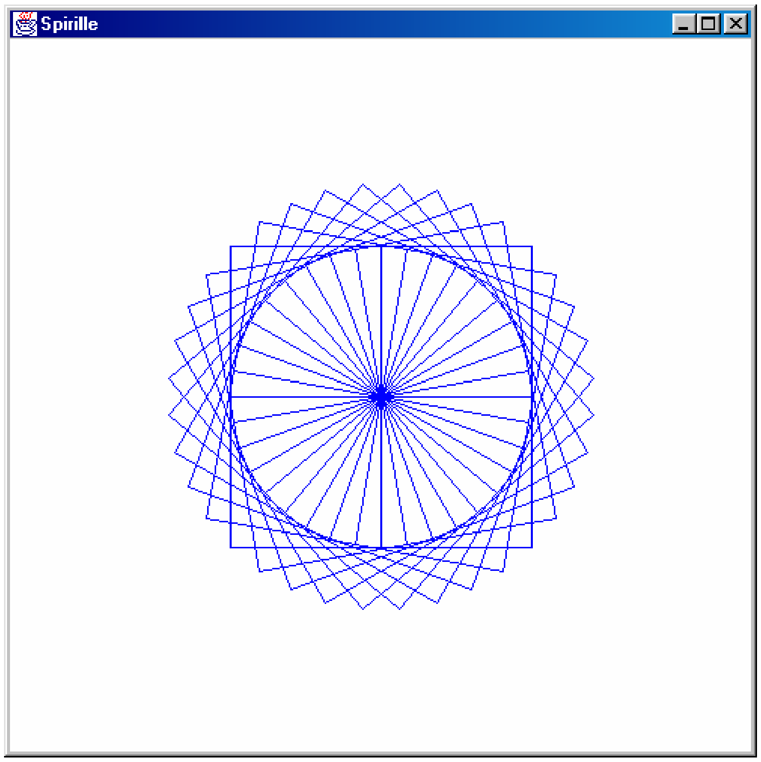


Fig. 2. The Spirille.

Program 1. The "Spirille" program

```
// 36 squares each turned an angle
// of 10 degrees from the previous
public class Spirille {

  public static void drawSpirille() {
    Turtle t = new Turtle();

    t.setColor(Color.blue);
    t.center();
    for (int i= 0; i<36; i++) {
      for (int j= 0; j<4; j++) {
        t.move(100);
        t.turn(90);
      }
    t.turn(10);
    }
  }
}
```

At this point, through a concrete and highly visual metaphor, students have already an intuitive first understanding of fundamental object-oriented concepts: object, object identification and message passing, as well as fundamental procedural concepts: state, flow of control (including loops) and parameters. The immediate visual feedback from the program makes it easy for students to identify logical programming errors and helps the inexperienced student; at the same time the material is still advanced enough to challenge those students that are already familiar with the basic topics.

The lab exercises are about making simple drawing (a flag and a house), nested drawings (pyramid seen from the top, a high-rise block, etc.) and animations (various objects that move around).

Typically, students can be divided into two groups; one group of students tend to use the relative commands turn and move whereas others are more comfortable with the absolute commands `turnTo` and `moveTo`. We discuss the different approaches in class, and in particular we investigate the difference of using the relative and the absolute commands. This turns into a discussion on important and fundamental software engineering issues such as generality, modifiability and reusability of programs.

5 Objects and Classes

A natural next step is to introduce two turtles into the same drawing area. This seemingly trivial addition is actually an intuitive and powerful way to introduce the students to another important range of fundamental concepts in object orientation—a trivial and natural step in an object-oriented language but difficult in the original turtle machine.

Having two turtles makes the importance of object identification clear: How else can you identify the actual turtle to whom a message is sent? Another reinforced point is that the two turtles have different states though they share a common behaviour—they appear and draw in different areas of the drawing area. From this example it is natural to discuss the benefits of categorising objects with common behaviour, and give examples from everyday life where we classify concepts and phenomena. Introducing the notion of a (Java) class is thus relatively easy.

6 Class Hierarchies and Procedural Abstraction

The next step is to introduce procedural abstraction through defining new methods to draw, say, a rectangle. At first sight this seems like an overwhelming task to do in the second lecture as the only way to add a new method in Java is either to introduce it into the Turtle implementation or to extend the Turtle class and introduce the method in the subclass. The first alternative is not an option—primarily because the turtle is provided as a Java package and secondly because we do not want to expose the implementation with all its details of the Java graphics. But the second option, to extend the Turtle class, turns out to be quite natural as described below.

6.1 Class Hierarchies

What do you do when you want your turtle to learn new “tricks”, say, drawing a rectangle? You train your turtle until its behaviour extends to include the ability to draw rectangles—and your turtle becomes a skilled turtle.

Program 2. Procedural abstraction and parameterisation

```
public class SkilledTurtle extends Turtle {

  public void rectangle(int w, int h) {
    for (int j= 0; j<4; j++) {
      t.move(100);
      t.turn(90);
    }
  }

  public static void main() {
    SkilledTurtle t= new SkilledTurtle();

    ... t.rectangle(100, 50); ...
  }
}
```

Program 3. Specialisation of turtles

```
public class GeometryTurtle extends Turtle {
  public void rectangle(int w, int h) { ... }
  public void circle(int r) { ... }
  ...
}

public class ArchitectTurtle extends GeometryTurtle {
  public void window(int w, int h) { ... }
  public void door(int w, int h) { ... }
  public void roof(int w, int h) { ... }
  ...
}
```

By focusing on the idea of ‘extending behaviour’ the Java syntax for declaring subclasses seems feasible (program 2 and 3). We show the students how (program 2), and they are able to mimic the idea in exercises where turtles with new special skills are required as exemplified in program 3. We do not dwell on abstract, complex, properties of inheritance and class hierarchies; rather, we show how this technique—grounded in an intuitive understanding of “training turtles”—can be used to solve a concrete problem. In this way we have an excellent basis for a thorough treatment later in the course when the students have concrete experience and an intuitive understanding of inheritance. Also, the students have seen an aspect of what inheritance is actually used for—and in the end we find this is the basic purpose of the course: not merely to understand language constructs and object oriented principles but being able to apply them to solve recurring problems in computer science.

6.2 Procedural Abstraction and Design by Contract

Based on the metaphor of skilled turtles the focus is turned to the problem of "training". The first skilled turtle is one that can draw rectangles, and clearly, one wants to be able to define once and for all how to draw a rectangle with width w and height h (program 2).

From the SkilledTurtle example (or similar ones) we initiate a discussion on the necessity of the last `t.turn(90)` in the procedure of program 2. The statement is superfluous as far as the resulting drawing is concerned, but there are obvious reasons to include the statement: to leave the turtle in the same state as before the call, making it easier to make composite drawings by multiple calls (like the Spirille). The students understand the point, and hopefully valuable seeds have been sown.

On the basis of simple examples like this we discuss important fundamental principles such as design, specifications and the distinction of what and how. In the context of the turtles, it comes natural for the students to express sound and well established principles for procedural abstractions, and later in the course when things get more complicated, we return to this common ground and recall the principles.

The moral of the discussion is that we need to be precise about what we want a piece of software to do. The best way to express such requirements is by writing a functional specification; hence we introduce the notion of design by contract [8], and from then on we use the technique throughout the course. This is reinforced as we provide the specification of the Turtle as JavaDoc API documentation, thereby forcing the students to become acquainted with the standard way of documenting Java classes and packages.

7 Recursion and Fractals

A traditional way to introduce recursion is to compute factorials. We find this unfortunate, because it introduces the technique on a problem for which it is inefficient and an iterative solution is straight-forward to express. Contrary to this, we introduce recursion for problems where the recursive solution is effective and iterative solutions are difficult to express elegantly.

The students are asked to write a program that can produce the list of drawings, Triangle, Penta and Poly, in figure 3 (and the next seven figures which are given equally odd names). However, first we demonstrate how to write methods for the first two drawings (program 4).

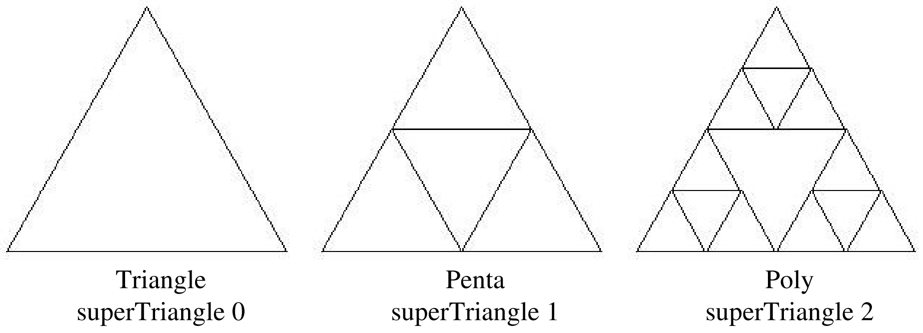


Fig. 3. superTriangles.

Program 4. Java code for triangle and penta

```
public class TriangleTurtle {
  public void triangle(int length) {
    for (int i= 0; i<3; i++) {
      move(length);
      turn(120);
    }
  }
  public void penta(int length) {
    triangle(length/2);
      move(length/2);
    triangle(length/2);
      turn(120); move(length/2); turn(-120);
    triangle(length/2);
      turn(-120); move(length/2); turn(120);
  }
}
```

As expected, the students produce eight new methods by copy-paste-and-substitute of the penta method. It works, but of course the students get the hunch that this cannot be the proper way to do it.

Once more we emphasise the notion of parameterisation, and we introduce the term `superTriangle(n)` to mean "a superTriangle of degree n". Defining `superTriangle(0)` to denote Triangle, superTriangle(1) to denote Penta and so forth, brings us more than half way towards the general solution; realizing that `superTriangle(-1)` does not make sense and handling this special case brings us the rest of the way (program 5).

Program 5. A general (recursive) solution

```
public class TriangleTurtle {
  public void triangle(int length) { ... }

  // pre: n ≥ 0
  public void superTriangle(int n, int length) {
    if ( n == 0 )
      triangle(length);
    else {
      superTriangle(n-1, length/2);
        move(length/2);
      superTriangle(n-1, length/2);
        turn(120); move(length/2); turn(-120);
      superTriangle(n-1, length/2);
        turn(-120); move(length/2); turn(120);
    }
  }
}
```

The derivation is fairly easy; with little guidance the derivation is almost exclusively done by the students. But even more interesting: Nobody mentions the notion of recursion; the solution just turns out to be what we call recursive.

8 Turtles as a Class Library

Later in the course,when we are covering more advanced object-oriented topics such as class hierarchies, polymorphism and application frameworks, we dig out the "old" Turtles package and use it as just another class library. We also .nd it important for students to use class libraries and the accompanying documentation as early as possible in the undergraduate curriculum, as pointed out in e.g. [14].

8.1 Class Hierarchies and Polymorphism

We use geometric shapes as example of a class hierarchy. An abstract class Shape has concrete methods move and erase and an abstract method draw that is implemented in subclasses of the Shape class. Each Shape instance has a turtle associated that it delegates the drawing tasks to; in this way the turtle becomes our graphical drawing library effectively encapsulating the Java specific graphical toolbox (figure 4).

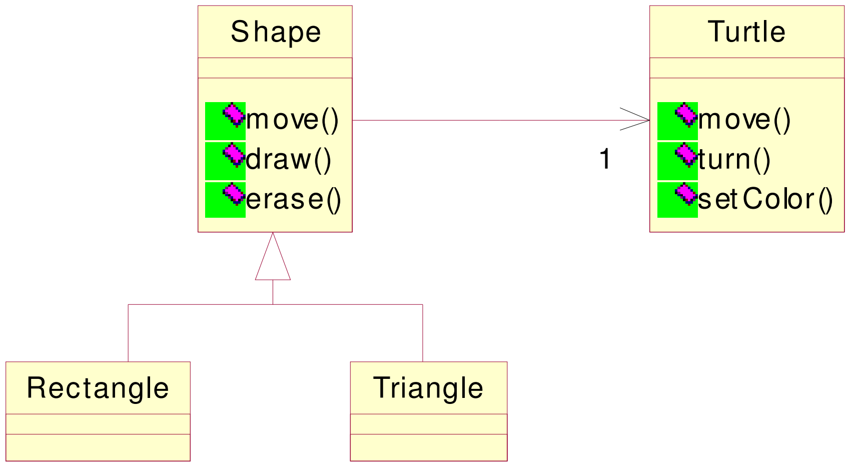


Fig. 4. A hierarchy of geometric shapes.

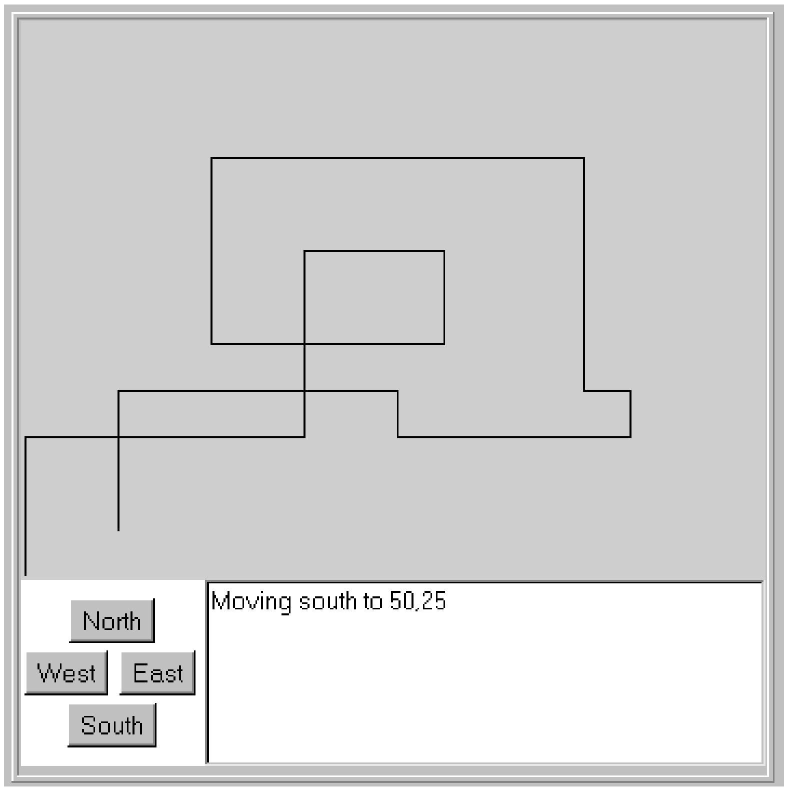
There is another important point in (re-)using the Turtles package as a drawing toolbox: Abstraction is the key concept in programming, and the code which is the intense focus of design, development, and testing today (the implementation view, how), will be taken for granted next month and simply used (the specification view, what). In a similar vein, the turtle was "the problem" in the beginning of the course—now it is the solution to the problem of drawing shapes in a new and different context.

8.2 Application Frameworks

Before introducing the students to GUI-programming with AWT or Swing, we give a lesson about frameworks in general, and we exemplify by providing a simple framework for the students. The purpose of the framework, called Presenter, is to allow fast development of graphical presentations of a set of images (actually graphical components) and text, where the ordering in the set is arranged using a familiar navigational metaphor: The compass with directions north, east, south, and west.

Our initial instantiation of the framework is a multimedia presentation of the tomb of Tutankhamun—using the compass buttons the user can move between the different chambers of the tomb, each chamber described both in text and by a picture from the original opening of the tomb.

In an exercise the students are asked to program a turtle controller i.e. the buttons North, West, South and East must control the movement of the turtle (moving at right angles),as shown in figure 5. While it shows the turtle in yet another context the main point here is that the turtle's drawing area is actually a subclass of java.awt.Component, the basic graphical component in Java, and therefore the framework accepts to display the turtle drawing area. This way another important property of inheritance is demonstrated to the students; not as much as a language construct, but as a technique for solving a specific set of problems.



Moving south to 50,25

Fig. 5. A turtle in the Presenter framework

9 Conclusion

We have described our use of a Java package, Turtles, which is an object-oriented variant of the classical turtle machine. Early in the course we are using the Turtles package to jump start our CS1 course by giving an intuitive introduction to classical procedural concepts in the spirit of the Logo language, introducing only the most necessary constructs and only by example; we do not want to provide detailed explanations that will not be understood nor remembered at this early stage.

Turtles is a great way to introduce simple as well as more advanced object-oriented concepts such as state, behaviour, object identification, inheritance, and polymorphism because the metaphor of a turtle on a drawing area is inherently an object-oriented model.

Furthermore, the Turtles package has been successfully used to illustrate abstraction at a later stage in the course: while the semantics and details of turtles were the focus and problems in the early part of the course, it is simply used as a drawing class library in the later part of the course.

The applicability of the Turtle graphics in introductory programming is acknowledged by the ACM Java Task Force who have included a class GTurtle in the acm.graphics package of the JTF library [13].

Though we have not conducted qualitative nor quantitative analysis of the effectiveness of our use of turtles to introduce object-oriented concepts to students, we have many indications of the positive effect. Our teaching assistants report that most students are proficient in basic object-oriented and procedural techniques early in the course, and students report using the turtles as fun and motivational. After all, this is not too bad.

9 Acknowledgement

We acknowledge Jens Bennedsen for stimulating discussions and collaboration during early development and use of the Turtle Machine.

References

1. Abelson, H., and diSessa, H. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press, 1980.
2. ACM SIGCSE. *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* (March 1993), vol. 25 of SIGCSE Bulletin.
3. ACM SIGCSE. *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* (March 1995), vol. 27 of SIGCSE Bulletin.
4. Astrachan, O., and Reed, D. “The Applied Apprenticeship Approach to CS1”, in *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* [3].
5. Brightman, H.J. *On Learning Styles*. Technical Report, Georgia State University, 1998. www.gsu.edu/~dschjb/masterteacher.html.
6. Decker, R., and Hirshfield, S. “Top-Down Teaching: Object-Oriented Programming in CS 1”, in *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* [2].
7. Hilburn, T.B. “A Top-Down Approach to Teaching an Introductory Computer Science Course”, in *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* [2].
8. Meyer, B. *Object-Oriented Software Construction* (2nd edition). Prentice-Hall, 1997.
9. Myers, I.B. and McCaulley, M. *Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator*, Consulting Psychologist Press, 1985.
10. Papert, S. *Children, Computers, and Powerful Ideas*, Harvester Press, 1980.
11. Pattis, R.E. “The ‘Procedures Early’ Approach in CS 1: A Heresy”, in *The Papers of the Twentyfourth SIGCSE Technical Symposium on Computer Science Education* [2], pp. 122–126.

12. Reek, M. “A Top-Down Approach to Teaching Programming”, in *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* [3], pp. 6–9.
13. Roberts, E. et al. “The ACM Java Task Force Version 1.0”, http://jtf.acm.org/.
14. Tewari, R. and Gitlin, D. “On Object-Oriented Libraries in the Undergraduate Curriculum: Importance and Effectiveness”, in *The Papers of the Twenty-fifth SIGCSE Technical Symposium on Computer Science Education* (March 1994), vol. 26 of SIGCSE Bulletin, ACM SIGCSE, pp. 319–323.

18 Frameworks in CS1

The paper *Frameworks in CS1: a different way of introducing event-driven programming* presented in this chapter has been published as a conference paper [Christensen et al. 2002] and as a chapter [Christensen et al. 2007] of the forthcoming book [Bennedsen et al. 2007a].

The book chapter is a revised version of the conference paper. The content of this chapter is equal to the conference paper [Christensen et al. 2002].

[Bennedsen et al. 2007a] Bennedsen, J., Caspersen, M.E. and Kölling, M., (Eds.) *Reflections on the Teaching of Programming.* Springer-Verlag, 2007.

[Christensen et al. 2002] Christensen, H.B. and Caspersen, M.E., “Frameworks in CS1: a different way of introducing event-driven programming”, *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education,* Aarhus, Denmark, pp. 75-79, 2002.

[Christensen et al. 2007] Christensen, H.B. and Caspersen, M.E., “Frameworks and their Role in Teaching”. In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007.

Frameworks in CS1 – a Different Way of Introducing Event-driven Programming

Henrik Bærbak Christensen
Department of Computer Science
University of Aarhus
8200 Aarhus N, Denmark
hbc@daimi.au.dk

Michael E. Caspersen
Department of Computer Science
University of Aarhus
8200 Aarhus N, Denmark
mec@daimi.au.dk

ABSTRACT

In this paper we argue that introducing *object-oriented frameworks* as subject already in the CS1 curriculum is important if we are to train the programmers of tomorrow to become just as much *software reusers* as *software producers*. We present a simple, graphical, framework that we have successfully used to introduce the principles of object-oriented frameworks to students at the introductory programming level. Our framework, while simple, introduces central abstractions such as *inversion of control*, *event-driven programming*, and *variability points/hot-spots*. This has provided a good starting point for introducing graphical user interface frameworks such as Java Swing and AWT as the students are not overwhelmed by all the details of such frameworks right away but given a conceptual road-map and practical experience that allow them to cope with the complexity.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software; I.3 [**Computing Methodologies**]: Computer Graphics; K.3 [**Computing Milieux**]: Computers and Education

General Terms

Design, Human Factors

Keywords

CS 1 Curriculum, Event-driven Programming, Frameworks

1. INTRODUCTION

We are presently teaching a CS1 course with an objects-first approach using Java. The curriculum covers four central subjects:

- Jump start: classes, objects, methods, control flow, parameterization
- Basic object-oriented programming: state, behavior, information hiding, modeling, UML to Java
- Algorithmic patterns: sweep, loop invariants, searching, merging, divide-and-conquer
- Advanced object-oriented programming: polymorphism, interfaces, specifications, design-by-contract, class invariants, frameworks, GUI-programming

As part of the advanced object-oriented programming subject, we teach the principles of programming using *frameworks* (not building them) and, based upon this treatment, graphical user interface frameworks. While the subject of frameworks is considered an advanced and complex topic, we have decided to include it in the CS1 curriculum nevertheless for several reasons.

The most important reason is the realization that the programming context most programmers are facing today is radically different from the one that existed, say, 10-15 years ago. Few modern programs (or "systems") are monolithic entities; instead they draw upon functionality from many different sources: third party components, object-oriented frameworks, dynamic link libraries, etc. Thus programming today is more a matter of "gluing" application code together with one or several frameworks and components than to be able to write a large monolithic program that does it all by itself. In this changed context we find it natural that students at an early stage are exposed to sound principles for making their code cooperate with third party code, and we find object-oriented frameworks to be a good vehicle for demonstrating these principles.

Another reason is pinpointed by Culwin [6]—today's students are confronted exclusively with graphical user interfaces as the interface to programs. To demonstrate programming only using text IO is not very motivating.

We also strongly think that we as teachers have a prime responsibility to educate *software reusers* just as much as *software producers*. Systematic software reuse [8, 10] is at the moment our best cure against the "software crisis". Forcing students to program using frameworks is the right step towards producing software reusers—and the result of their efforts look much more professional than mere sequences of, say, prime numbers in a text box.

In this paper, we describe the introductory framework and the exercises associated with it. We then discuss how the terminology introduced by the simple framework is used in our introduction to Java AWT as representative of modern object-oriented graphical user interface frameworks. Finally we summarize and discuss some of our experiences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'02, June 24-26, 2002, Aarhus, Denmark.
Copyright 2002 ACM 1-58113-499-1/02/0006 ...$5.00.

2. A TWO-STEP LEARNING PROCESS

We have adopted a *two-step* learning process for introducing graphical user interface frameworks in order to lessen the learning curve [3].

In the first step we teach the students a basic understanding of the principles underlying frameworks, using a concrete, simple, yet flexible, framework example. The example has nevertheless the fundamental characteristics of a framework:

- *Inversion of control*: The framework defines the control flow and collaboration patterns of the objects in the final application, instead of the usual "driver" program that the students write themselves.

- *Hotspots* [11]: The provided framework is abstract and needs to be specialized to the particular domain of the final application. Abstract classes that must be subclassed define the hotspots of our concrete framework.

In the second step we introduce Java AWT to the students through the context and terminology introduced by the first step—what are the hotspots of AWT and how do we tailor them to our needs? The main point is that AWT/Swing is large and complicated and thus confusing to the beginner, and you simply must master the underlying concepts and principles in order not to be overwhelmed by the sheer number of classes and methods.

3. FIRST STEP: PRESENTER FRAMEWORK

Our requirements of the framework were the following aspects: It should illustrate the basic principles of frameworks (inversion of control and hotspots); it should be simple for students to use; it should be flexible in the sense that a number of sensible instantiations should be possible; it should be fun, challenging, and visual.

The result is a *presenter framework*. The presenter framework facilitates construction of multi-media presentations of a domain where the compass-directions are a suitable metaphor for user navigation. (So far "multi-media" is limited to images and text but it is straightforward to extend it to movies and sound.)

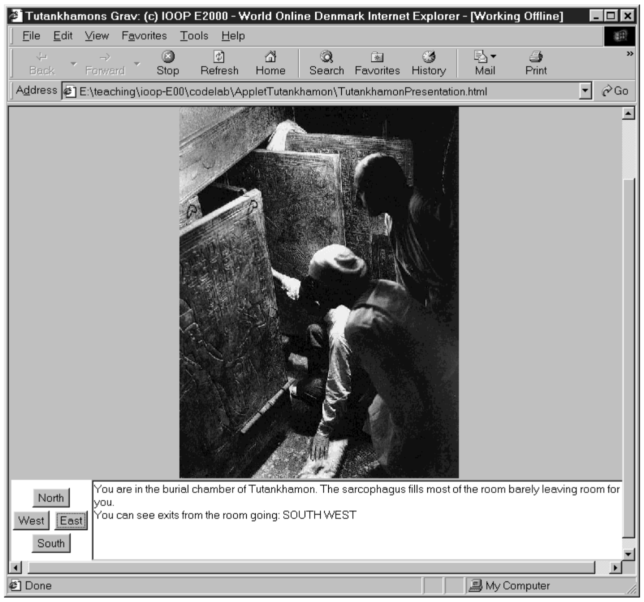
In our *first step* lecture we introduce the presenter framework through a specific instantiation, namely a multi-media presentation of the tomb of Tutankhamen, the pharaoh whose tomb was miraculously found rather intact in 1922 by Howard Carter [4].

In fig. 1 is shown a screen snapshot of the Tutankhamen tomb presentation. The presenter framework is an applet thus the presentation and later the student exercises can be run in a web browser.

Using the four buttons marked with the compass directions the user can navigate around the chambers of the tomb. In each chamber the user is presented with a picture taken during the original opening of the tomb along with some explanatory text.

It is our experience that the concrete instantiation—moving around a tomb with pictures from the original opening—grabs the imagination of the students.

The Tutankhamen's tomb instantiation also allows us to underline an important software engineering principle, namely separating model/domain code and user interaction code. We build a small object-oriented model of the domain with classes: *chamber* (having exits, an image and a description) and *visitor* (having an association with a specific chamber and a `move` method). As the user interaction code is completely defined by the framework, it is simply impossible for the students to mix UI and model code except through the well-defined hotspots provided by the framework.


Tutankhamons Grav: (c) IOOP E2000 - World Online Denmark Internet Explorer - [Working Offline]
File
Edit
View
Favorites
Tools
Help
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Mail
Print
Address
E:\teaching\ioop-E00\codelab\AppletTutankhamon\TutankhamonPresentation.html
Go
North
West
East
South
Done
My Computer

You are in the burial chamber of Tutankhamon. The sarcophagus fills most of the room barely leaving room for you.
You can see exits from the room going: SOUTH WEST

Figure 1: The presenter framework instantiated to present Tutankhamen's tomb.

3.1 Design

The presenter framework provides the application programmer with a simple interface (in practice the interface is split into two, as described in the next section):

```
public abstract class ImagePresenter
{
 public void showImage(String filename)
 {...}
 public void showText(String text) {...}

 public abstract void northButtonPressed();
 public abstract void eastButtonPressed();
 public abstract void southButtonPressed();
 public abstract void westButtonPressed();
}
```

An instance of `ImagePresenter` is an applet that provides the graphical user interface: a large area for displaying images, a smaller one for displaying text, and the four compass direction buttons that respond to user clicks.

The `showImage` and `showText` methods are methods that provide services for the application programmer (the students are well versed in object oriented thinking at this point in the course).

Thus, to instantiate the tomb presentation is a matter of overriding the `..ButtonPressed()` methods as e.g. in:

```
public void northButtonPressed() {
  visitor.move(NORTH);
}
```

where the move method of visitor must test for an exit leading north and invoke the `showImage` and `showText` methods with appropriate parameters.

The new technique the students must adopt is that in order to provide application specific functionality that reacts on user interaction, they have to subclass the abstract `ImagePresenter` to define the actions to perform when the user presses the buttons on the user interface. This raises discussions on the central points in frameworks as outlined below.

3.2 Inversion of control

In their previous programming experience from example code and exercises, there are always a number of interacting objects and a single 'driver' that does the setup and defines the main control flow. Now the control flow is dictated and controlled by the presenter framework instead. The application code comes into play only when the overridden `...ButtonPressed()` methods are called. This is a simple variant of event-driven programming and illustrates the inversion of control principle.

3.3 Hotspots

Frameworks define core functionality, control flow and object collaboration patterns. Application programmers refine frameworks to specific domains by adding code at well-defined points denoted hotspots (also called hooks or variability points). Hotspots can be defined using a number of different techniques: callback methods, objects that implement interfaces, subclassing, etc. We have adopted the subclassing technique as we find it the simplest and as it also demonstrates yet another use of polymorphism and specialization.

4. ELABORATION

We found that the framework could be used in more contexts by introducing a higher level of abstraction: A presenter that does not demand that the central graphical area is an image. Thus we split the framework into providing a `Presenter` class and a more specific subclass `ImagePresenter`, the latter being the one used for the tomb instantiation. The `Presenter` only demands that the graphical centre component is a Java AWT component and provides an abstract factory method [7] for subclasses to define the concrete instance.

Thus, the real framework classes are:

```
public abstract class Presenter
  extends java.applet.Applet
  implements ActionListener
{
 public abstract java.awt.Component
          createCenterComponent();
 public void showText(String text) {...}
 public abstract void northButtonPressed();
 public abstract void eastButtonPressed();
 public abstract void southButtonPressed();
 public abstract void westButtonPressed();
 ...
}
public abstract class ImagePresenter
       extends Presenter
{
 public void showImage(String filename){...}
 public Component createCenterComponent() {
   // return a Canvas instance
   // that can display images
 }
}
```

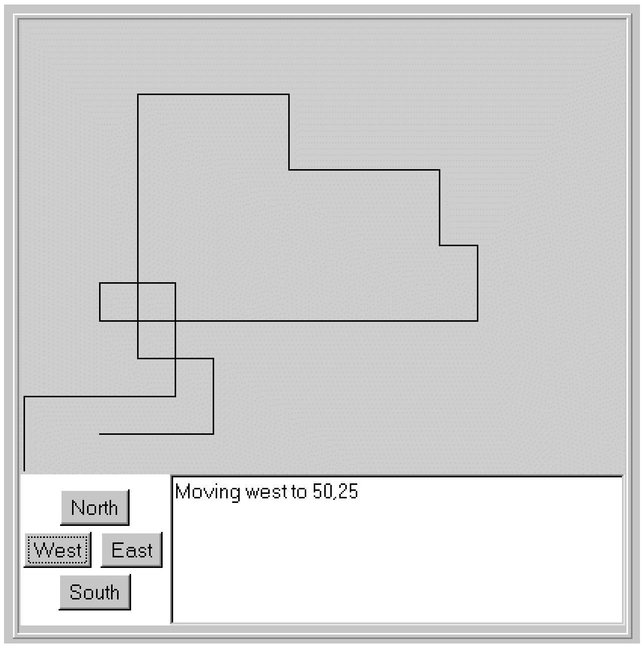
5. STUDENT EXERCISES

Several interesting, yet simple, instantiations can be made from the Presenter and ImagePresenter frameworks.

The first exercise is to make a virtual tour of a museum or gallery; a layout of a number of locations in a gallery is defined and a painting is associated with each location. The buttons can be used to move around the gallery and see the various paintings. This exercise is deliberately similar to the tomb instantiation. In another exercise only the "north" and "south" buttons are used to run through a list of images, essentially making the presenter a slide-show application.

The basic directional navigation metaphor also lends itself naturally to "classic" adventure games. We have an extension of the framework to include the ability to show two scrollable lists of images, one on either side of the center image. The application programmer can then program these so that one list represents an inventory of objects (images) carried by the user and the other list represents an inventory of objects in the visited location. A click-event on an image in a list is a hotspot of the framework that the student can refine to mean that objects are moved between the two inventories.

In other exercises we base ourselves on the *Presenter* class that takes any `java.awt.Component` as center component. Our course uses an object-oriented variant of turtle graphics to introduce people to programming and object-oriented thinking [5]. We therefore ask the students to make a demonstration of the turtle where the turtle moves some distance in the direction corresponding to the compass direction that the user clicks. A snapshot of the turtle instantiation is shown in fig. 2.



Moving west to 50,25

Figure 2: The presenter framework instantiated to demonstrate turtle graphics.

After having introduced the students to AWT, a slightly more advanced exercises in instantiating the presenter framework is to make a 4x4 slide-puzzle by defining a grid of buttons marked with the numbers 1–15 and an empty button denoting the "hole". The "hole" is then moved by pressing the compass buttons so the user can try and solve the puzzle by arranging the numbers in the right pattern in the grid.

In summary we find that though the provided functionality of the framework is limited and simple, there are a number of intriguing exercises to be made based upon the framework that forces the students to negotiate the basic principles of inversion of control and refining hotspots.

6. SECOND STEP: JAVA AWT

The next step in the learning is introducing a real GUI framework. We restrict ourselves to AWT instead of Swing: the principles are the same but Swing contains even more detail that may blur the picture for the students.

We have many indications that the students are helped by the presenter framework as they learn AWT. They have seen the inversion of control principle; they have seen the principle of refining hotspots and can now concentrate on the particular technique used in AWT for doing this refinement; finally, they are acquainted with the underlying concepts and principles of framework design.

7. EXPERIENCE

At the time of writing our CS1 course has been taught seven times. While we have made many changes in the course material over the years, the `Presenter` framework has been taught with success every time. As the framework has been used ever since we started teaching this course we have no comparative evaluations of the advantages and drawbacks of our approach compared to other ways of introducing graphical user interfaces. However, we have a number of experiences; though they are not rigid scientific evaluations, they do illustrate key aspects of our approach.

First of all the students generally value the approach: The exercises are reported as “fun” and not too hard, the students value the visual appearance of their programs, and most importantly they value that the framework terminology they have learned is used to ease and enhance their understanding of the much more complicated AWT.

At the exams the students demonstrate adequate performance on the topics of frameworks and graphical user interfaces, but of course it is very difficult to measure curriculum quality from exams.

Finally we also find that the impact of teaching frameworks must be measured on a long-term scale. We feel that even to students that “just don’t get it” at this early stage, we have still planted a small but important seed that will ease their learning of framework theory, reuse techniques, design patterns, and software architecture at a later stage.

8. RELATED WORK

To the best of our knowledge our approach is novel and has not been reported elsewhere. However, several authors have reported and discussed approaches for teaching how to program graphical user interfaces at an early point in the CS curriculum. Common to most approaches is the desire to shield students from the underlying complexity (through the use of design patterns such as adapter and wrapper) more than to provide the conceptual tools to understand the complexity.

Woodworth et al. [14] describe how migrating from console- to event-driven models can be eased by introducing a module that acts as an adapter between the event-driven user interface and the domain classes. This way the adapter behaves like the program driver the students are used to from the console driven model. Wolz et al. [13, 12] describe an approach that reduces the complexity of GUI programming by wrapping the underlying user interface toolkit in simpler abstractions.

Bruce et al. [2] describe an interesting approach where event-driven models are introduced right at the start of the course and report their approach to be successful.

Common to most approaches is that the main goal is to teach programming *GUI toolkits* in CS1 and the event-driven model is the obstacle to be handled (by wrapping it, adapting it, or otherwise simplifying it). Our focus is radically different. Our main goal is to teach *frameworks* and a GUI framework is just one type of framework (although an important one). Teaching frameworks is teaching inversion of control and how to refine hotspots, i.e. the event-driven model comes out as a special case of inherent framework behavior.

Buck et al. [3] outline an inside/out pedagogical approach based on Bloom’s taxonomy for cognitive development. We find that our two-step approach is in line with their ideas as the introductory framework has relatively simple building blocks that allow students to comprehend the basic concepts before they are asked to apply them to build GUI interfaces themselves.

Our approach is an instance of the *early bird pedagogical pattern* by Bergin [1]. We find that frameworks, reuse, and reuse techniques are extremely important and must be presented at an early point in the careers of the students and reinforced throughout their studies.

9. SUMMARY

We have described our two-step approach for teaching the principles of object-oriented frameworks. In the first step, we introduce a simple framework that nevertheless has all main features of a full-blown framework. This allows us to concentrate on the main principles underlying frameworks without distracting details. In the second step, we expose the students to the AWT framework but can now draw upon their experiences with the concepts from the much simpler `Presenter` framework.

As outlined earlier, we cannot present rigid evaluations that demonstrate the strength of our approach. We do feel, however, that our argumentation in favour of the approach is strong and valid. The learning curve to climb for the students in order to tackle object-oriented graphical user interface frameworks is a steep one, and breaking it into smaller steps is essential to succeed. Our approach is one of many possible ways of providing such smaller steps but it has some unique benefits. It focuses on fundamental issues in frameworks and reuse techniques instead of concentrating narrowly on event-driven user interfaces. The students are introduced to the principles of object-oriented frameworks. In the `Presenter` framework they are forced to separate domain model code from their user interaction code, which is accepted as a superior architecture for designing interactive applications. They are taught that a programmer of today reuses code provided by others instead of building everything from scratch. Finally they are taught some of the central techniques for integrating reusable code with their own application code laying a strong basis for later courses that teach design patterns and software architecture.

The `Presenter` framework and sample instantiations can be obtained free of charge by contacting one of the authors.

10. REFERENCES

[1] Bergin, J. Fourteen Pedagogical Patterns. http://www.csis.pace.edu/~bergin/PedPat1.3.html.

[2] Bruce, K. B., Danyluk, A. P., and Murtagh, T. P. Event-driven Programming is Simple Enough for CS1. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE’01* (Canterbury, UK, 2001), pp. 1–4.

[3] Buck, D., and Stucki, D. J. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education* (Austin, Texas, USA, mar 2000), pp. 75–79.

[4] Carter, H., Mace, A., and White, J. M. *The Discovery of the Tomb of Tutankhamen*. Dover Publications, 1985.

[5] Caspersen, M. E., and Christensen, H. B. Here, There and Everywhere — On the Recurring Use of Turtle Graphics in CS1. In *Proceedings of the Fourth Australasian Computing Education Conference, ACE 2000* (Melbourne, Australia, Dec 2000), pp. 34–49.

[6] Culwin, F. Object Imperatives. In Joyce [9], pp. 31–36.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[8] Jacobson, I., Griss, M., and Jonsson, P. *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press, 1997.

[9] Joyce, D., Ed. *Thirtieth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisianna, mar 1999).

[10] Karlsson, E.-A. *Software Reuse – A Holistic Approach*. John Wiley and Sons, 1995.

[11] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

[12] Wolz, U., and Koffman, E. simpleIO: a Java package for novice interactive and graphics programming. In *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education* (Krakow, Poland, jun 1999), pp. 139–142.

[13] Wolz, U., Weisgarber, S., Domen, D., and McAuliffe, M. Teaching introductory programming in the multi-media world. In *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE'96* (Barcelona, Spain, jun 1996), pp. 57–59.

[14] Woodworth, P., and Dann, W. Integrating Console and Event-Driven Models in CS1. In Joyce [9], pp. 132–135.

19 Model-Driven Programming

The paper *Model-Driven Programming* presented in this chapter has been published as a conference paper [Bennedsen et al. 2004] and as a chapter [Bennedsen et al. 2007c] of the forthcoming book [Bennedsen et al. 2007a].

The book chapter is a revised version of the conference paper. The content of this chapter is equal to the book chapter [Bennedsen et al. 2007c].

[Bennedsen et al. 2004] Bennedsen, J. and Caspersen, M.E., “Programming in context: A model-first approach to CS1”, *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, USA, pp. 477-481, 2004.

[Bennedsen et al. 2007a] Bennedsen, J., Caspersen, M.E. and Kölling, M., (Eds.) *Reflections on the Teaching of Programming.* Springer-Verlag, 2007.

[Bennedsen et al. 2007c] Bennedsen, J. and Caspersen, M.E., “Model-Driven Programming”. In *Reflections on the Teaching of Programming,* Springer-Verlag, 2007c.

Model-Driven Programming[1]

Jens Bennedsen* and Michael Caspersen#

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

#Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

Conceptual modelling is the defining characteristic of object-orientation and provides a unifying perspective and a pedagogical approach focusing upon the modelling aspects of object-orientation. Reinforcing conceptual modelling as a basis for CS1 provides a course structure integrate the core elements from a conceptual framework for object-orientation and a systematic approach to programming; both of these are help to newcomers. The progression of the course is defined by the growing complexity of the conceptual model which is to be implemented. The focus is not on conceptual modelling per se, but on the use of conceptual models as a structuring mechanism and a guide for the implementation. In this article we discuss different ways to structure an introductory programming course and give concrete examples on how a course where the complexity of the conceptual model is defining the structure.

1. INTRODUCTION

Over the years there have been ongoing discussions on the content and structure of an introductory programming course, what programming language and tools to use in such a course as well as the pedagogy to apply. There have been many suggestions (e.g. Bell & Scott 1987; Evans & Patterson 1985; Koffman & Wolz 1999; Oldham 2005; Shaffer 1986; Henze, Nejdl & Wolpers 1999; Fjuk, Berge, Bennedsen, & Caspersen, 2004). Most of the suggestions are structured according to the complexity of the programming language and are focused on the syntax of the programming

[1] This chapter is partly based on Bennedsen, J. and Caspersen, M: "Programming in Context – A Model-First Approach to CS1"Procedings of the 35th SIGCSE technical symposium on Computer Science Education, 2004. p. 477 - 481

language. In this chapter we will describe and discuss what we have found to be a useful structuring mechanism for an introductory programming course, namely the complexity of the class model to be implemented.

1.1. Three implementations of a programming-first curriculum

In order to define a common computer science curriculum, including an introductory programming course, ACM and IEEE established the Joint Task Force on Computing Curricula 2001. The charter was: "To review the Joint ACM and IEEE CS Computing Curricula 1991 and develop a revised and enhanced version for the year that will match the latest developments of computing technologies in the past decade and endure through the next decade". In the final report (The joint task force 2001), the role and place of programming in the curriculum is discussed. Is programming what needs to be taught first (what they call a programming-first approach) or are there other topics that need attention first? The conclusion is: "the programming-first model is likely to remain dominant for the foreseeable future". (p. 24)

The report describes three implementations of a programming-first curriculum based on three programming paradigms: The imperative, the functional and the object-oriented paradigm. The object-oriented paradigm has gained much interest in the past decade resulting in many textbooks (e.g. Arnow, Dexter & Weiss 2004; Barnes & Kölling 2003; Horstmann 2001; Niño & Hosch 2001) and much interest among teachers on implementing the object-first strategy (e.g. Alphonce & Ventura 2002; Cooper, Dann,& Pausch 2003).

1.2. Objects-first curriculum

Objects-first is not a well-defined term. It seems that every CS1 teacher has his or her own interpretation of the term (e.g. Cooper, Dann & Pausch 2003; Jones, Boyle & Pickard 2003; Kölling & Rosenberg 2001; Schmolitzky 2004). The Joint Task Force (2001) described objects-first as: "an objects-first approach that emphasizes early use of objects and object-oriented design" (p. 28). What does early mean, and what is meant by object-oriented design?

To add to the confusion of the objects-first concept is the problem that students often struggle with, namely the concepts "class" and "object". They see and work with the program text and therefore mostly with the classes and not object directly. To have students understand the difference between the class as the concept on compile time and the objects on run-time is a major challenge (Fleury, 2000; Holland, Griffiths, & Woodman, 1997) We have experienced that the explicit use of the conceptual framework for object-orientation and talking about it helps the students to understand it (for example by talking about a concepts extension, intension and designation (Madsen, Møller-Petersen & Nygaard 1993 p. 291) and to use a tool that supports intuitive and easy creation of objects from the cases (Kölling 2005))

Lewis (2000) discusses nine myths about object-orientation and its pedagogy; one is that the phrase "objects first" is well defined. The author writes: "No matter what your definition of objects first is, it is likely to be different from that of the person next to you." (p. 247), and "The phrases 'objects first' and 'objects early' are bandied about in a variety of contexts. When discussing a CSI course they are often used to convey the general idea that objects are discussed early in the course and established as a fundamental concept. Beyond that, however, these phrases seem to take on a variety of meanings, with important implications." (p. 246).

Our definition of objects-first is:

- Objects from day one – in the beginning the students uses predefined classes to create objects, then they imitate the implementation of a class and finally they creates classes.
- A balanced view on the three perspectives on the role of a programming language (see next section)
- Enforcing the use of a systematic way to implement a description of a solution (see section 1.4 Contracts)

1.3. The role of the programming language

In (Knudsen & Madsen 1988) three perspectives on the role of a programming language are described:

Instructing the computer: The programming language is viewed as a high-level machine language. The focus is on aspects of program execution such as storage layout, control flow and persistence. In the following we also refer to this perspective as coding.

Managing the program description: The programming language is used for an overview and understanding of the entire program. The focus is on aspects such as visibility, encapsulation, modularity, separate compilation.

Conceptual modelling: The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena.

These represent a widespread three-level perspective on object-oriented programming as represented by the three abstraction levels for the interpretation of UML (Rumbaugh, Jacobson & Booch 2005) class models (Fowler 2000): conceptual level, specification level and code/implementation level.

When designing a programming course one decides how much time, effort and focus are given to each of the three perspectives. It is possible just to focus on the first, instructing the computer, and ignore the two others. This results in a course where the details of the programming language are in focus but where the students do not learn the underlying programming paradigm. If on the other hand one just focuses on conceptual modelling (using a case-tool to generate code), the result is a course where the students cannot produce code by themselves. We find it vital to balance the three views on the role of the programming language. The primary advantages are

- A systematic approach to programming
- A deeper understanding of the programming process
- Focus on general programming concepts instead of language constructs in a particular programming language.

Most of the descriptions and discussions of the object-first strategy tend to focus on instructing the computer and managing the program description (see e.g. Rountree, Rountree & Robins 2003). To our knowledge, no introductory programming textbook exists that addresses conceptual modelling, and we have been able to find only a few articles discussing the adoption of conceptual modelling in CS1 (e.g. Alphonce & Ventura 2002; Knudsen & Madsen, 1996; Sicilia, 2006). It is our experience from many years of teaching CS1, that the inclusion of conceptual modelling perspective has a major impact on the students' skills and their understanding of the programming process. It is our conviction that the general omission of conceptual modelling is one of the major reasons for the problems identified by The Joint Task Force (2001, p. 23):"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 problems and problem-solving contexts in the future."

The Joint Task Force (2001) generally ignores conceptual modelling in the object-first recommendations for CS1. Aspects of conceptual modelling are mentioned only briefly and the recommended time to be used on the subject is four core hours!

1.4. Contracts

We identify contracts (Meyer 1992) and techniques for the systematic creation of object-oriented programs at four (six) different levels of abstraction:

1. *Problem domain → **conceptual model***: Create a UML class model of the problem domain, focusing on classes and structure between classes
2. *Problem domain → **Dynamic model***: Create a UML state chart to capture dynamic behaviour
3. *Conceptual model and dynamic model → **specification model***: Specify properties and distribute responsibility among classes.
4. *Specification model → **implementation***:
    a. *Specification model → **implementation of inter-class structure**:* Create a skeleton for the program using standard coding patterns for the different relations between classes.
    b. *Specification model → **implementation of intra-class structure**:* Create class invariants describing the internal constraints that have to be fulfilled before and after each method call.
    c. *Specification model → **implementation of methods**:* Use algorithm patterns for the traditional algorithmic problems e.g. sweeping, searching. Use loop-invariants for the systematic construction of loops.

In the introductory programming course focus is on the fourth level; beginning students cannot design (Pattis 1993), and therefore we provide a conceptual model/specification model as the basis of almost every programming assignment in the course.

We reinforce the notion of contracts at each level.

- At the conceptual level the contract is expressed as relations between classes; this contract is between the use and the programmer.

- At the specification level the contract is expressed as functional specifications of the interfaces (classes) in the model; this contract is between clients and implementations of interfaces.
- At the implementation level the contract is expressed as assertions in the program text (e.g. general assertions, class invariants, and loop invariants).

In the intro course we focus on contracts at the conceptual level and the implication of these contracts for the implementation in Java. It is our experience that the notion of contract in the context of a model-driven approach is a great help to beginning students.

2. CONCEPTUAL MODELING

In (Madsen, Møller-Petersen & Nygaard 1993) object-oriented programming is defined as follows:

A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world.

The key point here is model. An object-oriented program is a model, and this model can be viewed at different levels of detail characterized by different degrees of formality: An informal conceptual model describing key concepts from the problem domain and their relations, a more detailed class model giving a more detailed overview of the solution, and the actual implementation in an object-oriented programming language.

Object-orientation has a strong conceptual framework (notions of concepts and phenomena, identification of objects, identification of classes, classification, generalization and specialization, multiple classification, reference- and part-of composition). One of the advantages of the conceptual framework is that it gives an integrating perspective on analysis, design and programming thus making it much easier for the students to understand these normally fuzzy concepts. Analysis is the process by which you create a conceptual model of the problem domain, design is the process where you fit the model to the restrictions of the particular programming language and implementation environment, and implementation is coding the design model. Omitting this integrating perspective and focusing only on object-orientation for implementation will leave out one of the most important assets of object-orientation.

We focus on the conceptual modelling perspective, emphasizing that object-orientation is not merely a bag of solutions and technology, but a way to understand, describe and communicate about a problem domain and a concrete implementation of that domain.

The integration of conceptual modelling and coding provides structure, traceability and a systematic approach to program development which strongly motivates and supports the students in their understanding and practice of the programming process.

The course is still a programming course, not a course on how to create conceptual models of given phenomena. We do not expect the students to create conceptual models of the referent system (Madsen et al. 1993 p. 286), that is to perform the activity they call analysis (ibid p. 310) nor do we expect the students in the beginning to be able to create a design for the program. We supply them with the program design described as a class model, and then they implement this design in a systematic way.

3. STRUCTURE OF A MODEL-FIRST COURSE

In this section we discuss different aspect of a model-first programming course: Progression, goals, on example of a model-first course describing the progression in terms of the concepts from the object-oriented conceptual framework.

3.5. Progression

One of the key problems in designing a programming course is to define the progression – what to start with, what next and so on. Traditionally one starts with the simple things, but that quickly rises the question is: "Simple related to what?" Traditionally the answer is "Simple programming language constructs" i.e. the progression is defined by the complexity of the programming language. Robbins, Rountree and Rountree (2003) conclude that "typical introductory programming textbooks devote most of their content to presenting knowledge about a particular programming language" (pp 141), that is to say the majority of them are based on an "instructing the computer" or "managing the computer" perspective.

The approach taken here is to use the three perspectives on the role of the programming language as a guide for the structure of the course. In the first half of the course, roughly speaking, focus is concurrently on understanding and using a conceptual model as a blue print for programming and actual coding; in the second half of the course the primary focus is on internal software quality, i.e. managing the program description. The answer to the question on "simple related to what" is therefore simple related to the complexity of the underlying conceptual model.

In section 3.7 "A concrete implementation of a model-first course" our course design is presented. The interpretation is what we find simple and complex in the object-oriented conceptual framework; other interpretations are of cause doable. If you e.g. find inheritance simpler than association, then the next concept to introduce after the class concept is inheritance.

Apart from using the complexity of the underlying conceptual framework as a definition of progress we also use the "early bird" pedagogical pattern (Bergin): "*The course is organized so that the most important topics are taught first. Teach the most important material, the "big ideas," first (and often). When this seems impossible, teach the most important material as early as possible.*

3.6. Goals

Coding and understanding conceptual models is done hand-in-hand, with the latter leading the way. Introduction of the different language constructs are subordinate to the needs for implementing a given concept in the conceptual framework. After introducing a concept from the conceptual framework a corresponding coding pattern is introduced; a coding pattern is a guideline for the translation from UML to code of an element from the conceptual framework.

This approach supports a spiral course layout (Bergin), reinforcing the most important concepts several times in the course. There are two criteria for the design of the spiral layout: the most common concepts of the conceptual framework are introduced first, and throughout the course the students must be able to create working programs.

The conceptual framework is comprehensive; for CS1 we restrict the coverage to association, composition and specialization which by far are the most used concepts in object oriented modelling and programming.

The starting point is a class and properties of that class and the relationship between the class and the objects created from this class. One of the properties of a class can be an association to another class; consequently the next topic is association. This correlates nicely to the fact that association (reference) is the most common structure between classes (objects). Composition is a special case of association; composition is taught in the next round of the spiral. The last structure to be thoroughly covered is specialization. Specialization bridges nicely to the second half of the course where the focus is on software quality and design where specialization is often used as a way to make more flexible designs.

3.7. A concrete implementation of a model-first course

In the following subsections we describe some of the elements of the design of the course focusing on the first half of the course where transformation from models to code dominates.

Experience

The presentation below is based on the authors experience for more than 15 years of teaching introductory programming. The ideas have been used both in traditional university courses (5-10 ECTS) with a lecture style of teaching and 200+ students attending and in classroom style of teaching. It has been used both for young students just entered the university without any computer science background as well as adults in further education courses where the students have been programming in another paradigm (the imperative). It has been used for students majoring in a computer science as well as students with a liberal arts and humanities background.

Getting Started

We want to give the students an everyday understanding of object-orientation and a very informal understanding of the process of creating a UML class model. We therefore start by illustrating the concepts using everyday life situations in a role-play. The goal for the role-play is to illustrate structure and dynamics in terms of concepts, phenomena and messages in a problem domain and classes, objects and method calls in a corresponding (class and program) model. We use UML (primarily class diagrams) to describe concepts and their properties, without any formal introduction to the modelling language.

To introduce the students to basic coding we use a graphics package (Christensen & Caspersen 2000). The graphics package is presented in terms of a class diagram; hence, the students experience very early the strength of a class model as an abstract description of a program component as well as a communication tool; the UML-model provides an effective “language” for documenting and communicating about classes.

This introductory part of the course provides an external view of classes and objects. For a further discussion on these problems, see (Caspersen & Christensen “CS1: Getting started”, this volume).

Class

After having used classes and objects, we turn to an internal view and start writing classes; we do this by introducing the first coding pattern: Implementation of a class. A coding pattern is a general description of (one way of) implementation of an element of the conceptual framework. The students discuss a domain concept, select a few properties, and express the domain concept using UML. We emphasize however that the description of concept in it self is not important, we use that fact that the students themselves create the concept as a motivation for the students. Using the coding pattern the UML-description is systematically translated into Java code. The general coding pattern are not show to the students; the students observe implementations of classes and little by little abstract over these different implementations of different models. The learning is from Diverging, accommodating, converging to assimilating in the four learning styles described by Kolb (1984).

In this phase of the course the students learn about basic language constructs such as assignment, parameters, conditional statements; constructs needed for the systematic translation of model into code like classes and objects, state and behaviour, primitive types and object types, reference, parameterization, this, methods, attributes and constructors.

As described before a spiral approach is used. This implies that, for example, in the coverage of primitive types only what is strictly needed is taught. In this case we only use `int` and do not worry about the other types – they will be introduced when they are needed by the exercises.

The focus is on a systematic way of programming. This implies three things: many examples are shown to the students; explicit use of UML and a focus on the programming process (see Caspersen & Kölling, 2006 for details on the process)

The examples used are general concepts from the students every day life like Person, Account, Die and Date.

We use BlueJ (Kölling, Quig, Patterson & Rosenberg 2003, Kölling this volume) as the programming tool. In BlueJ the user has a kind of UML diagram, but the internal details of the classes are not shown. We therefore use drawings of a UML class in order to explain an abstract understanding of a class.

Since we find it important to focus on the programming process and not just the end products (the program), we use a lot of “live coding” (Hyland & Clynch 2002). The purpose of this is not to show the students the nice and linear way from problem to solution, but to show the students how a professional programmer attacks the problem, making the actions visible and a source of identification (Nielsen & Kvale

1997). For more elaboration on this see Caspersen & Bennedsen, The Programming Process this volume.

Association

In the model of the problem domain the most common structure between classes is an association. We use several examples with progressive complexity to illustrate the concept and its implementation.

One Class with a Reference to Itself

Through a number of progressive examples we illustrate that an association is a property of a class, a class can have more than one association, and an association is a dynamic relation.

The students extend a previous example with a recursive association. One example is that a Person can be married_to another Person or the lover of another Person. This results in the model in figure 1.

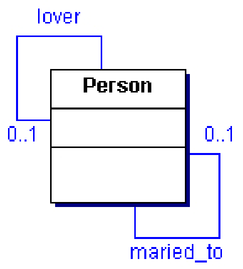


Figure 1: One class with two associations

In order to implement associations with 0..1 cardinality the student needs to know about programming language elements (e.g. reference and the null value). It also gives the students an understanding of interaction between objects (calling methods on other objects) and reference semantics.

Another example of a recursive association is a simple adventure game where the rooms in the game are connected to other rooms in different directions. This can be modelled by the following model:

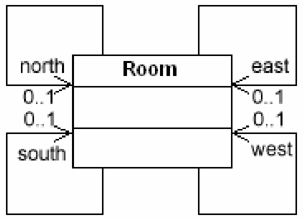


Figure 2: One class with four recursive associations

Again the idea is that the students sees many implementations of the same general concept from the conceptual framework and realizes the general coding pattern:

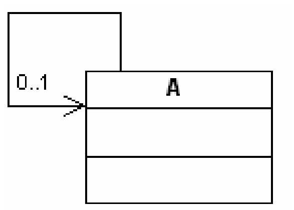


Figure 3: General, recursive association

This can be implemented using the following coding pattern:

```
public class A {

   private A a;

   public void setA(A a) {

      this.a=a;

   }

   public A getA() {

      return A;

   }

}
```

Turning to 0..* associations imply that the student needs to know about Collections (either one of the Java standard Collections or the array type) and the need for iteration arises (the for-each loop, an `Iterator` or an index variable and a simple loop). This is done using a simple algorithm pattern for sweeping through a collection. One example we use is the concepts of “friends” - a Person can have many friends:

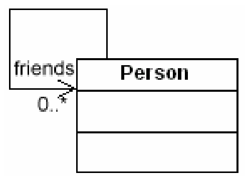


Figure 4: A person with many friends

More Classes

In order to get more interesting collaboration between classes, the next concept is associations between different classes. As a starting point we use a domain model with the following structure:

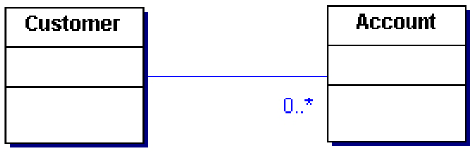


Figure 5: One customer can have many accounts

The students quickly understand that an association between different classes in principle is the same as a recursive association. This is true for the implementation as well; again the students generalize to a generic coding pattern for 0..* associations.

3.8. Composition, Specialization and Interfaces

We treat the remaining elements of the conceptual framework, composition and specialization, in a similar way. As mentioned earlier, specialization bridges nicely to the second half of the course focusing on software design and quality. The primary quality aspect is coupling and the main language construct by which to achieve low coupling is interfaces. Interfaces play an important role in the separation of specification and implementation: the specification of properties of a domain concept and (different) implementation(s) of these properties.

4. ON THE ROLE OF CONCEPTUAL MODELING IN CS1

In the following we will discuss some of the aspects of our integration of conceptual modelling in an introductory programming course. As mentioned in section 3.6 “Goals” it is not a goal in this course that the students creates conceptual models by themselves, but use the conceptual model as a map of the code guiding their actual programming.

4.9. Systematic Approach to Programming

The goal is to teach the students to appreciate and achieve quality software. By good quality software we mean modifiable software, i.e. readable and understandable programs with a good structure, low coupling and high cohesion. These quality measures are by no means obvious to newcomers, and how to achieve them is even harder. We need to teach the students guidelines for achieving it and a vocabulary to talk abut their programs in order to help them build quality programs. The guidelines can be at different levels – see section 1.4 “Contracts”

4.10. Providing Confidence

To program is difficult! In McCracken et al (2001) the authors found “shockingly low performance on simple programming problems, even among second-year, college-

level students at four schools in three different countries". It requires knowledge and skills of many things such as the programming language, development tools and the capability of formulating a solution in such a way that a computer is able to understand it. Especially the last demand implies the need for creativity when programming.
Students find the creative process very difficult. In a more traditional programming course students are guided by standard algorithmic techniques such as searching, sorting, divide and conquer etc. The problem is that algorithmic techniques do not help the students to create the overall structure of a solution; they do not know where and how to start because the mental gap between the problem description and an implementation in terms of algorithms is too big. Conceptual modelling gives a systematic and structured approach to programming which provides confidence and a safe ground for addressing the programming task.
Most programming tasks are trivial and can be handled using simple standard techniques such as the generic coding patterns described above. By focusing on standard techniques first, the need for algorithmic creativity is reduced (and a thorough treatment is postponed to CS2).

4.11. The Programming Process

The modelling approach to programming invites for an iterative process where the program is developed incrementally. Through progressive exercises we reinforce such a process in order to imitate modern program development processes (Beck 2000).

4.12. Abstraction

One of the important skills we want our students to possess is the capability to abstract. One way of stimulating the student's ability to abstract is to give several exercises with similar structure.

One example from the bank domain is the model shown in Figure 5. In a student administration domain we have the following model:

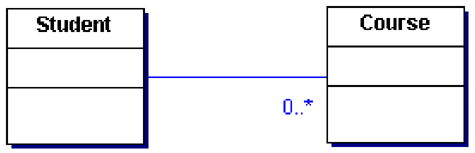


Figure 6: A student can participate in many courses

Initially the students see these two models as completely different, but gradually they realize they are both instantiations of the same abstract model:

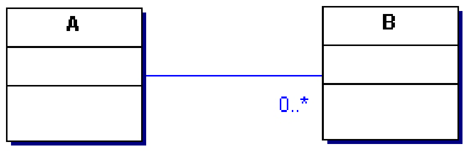


Figure 7: Abstract to many association

From this abstract model they can produce a corresponding generic coding pattern (see figure 8).

```
import java.util.*;

public class A

{   private Collection bs;

    public A()

    { bs = new ArrayList(); }

    public Collection getBs()

    { return bs; }

    public void addB(B b)
```

Figure 8: Generic coding pattern for 0..* association

4.13. Object-orientation and procedural programming

A part from implementing the overall static structure of a program, students need to implement the inside of the methods as well. As described in the section 1.4 "Contracts" we use several systematic approaches to this. In the introductory programming course however, we do not teach the complete picture of systematic tools useful for implementing methods nor does the students themselves create class invariants or loop invariants – we supply them in an informal way (e.g. by describing the role and constraints on each attribute of the classes using a comment or by general comments in the javadoc of the class) and shows the students how we as experienced programmers use this information when implementing methods. In a later course the students will learn how to create contracts them selves at all the levels mentioned in the section 1.4 "Contracts".

This focus on the use of contracts implies that our focus on the more traditional procedural aspects of programming is scaled back – the students learn how to implement general sweep algorithms but the more subtle problems related to algorithmic problem solving is postponed to a later course.

5. CONCLUSIONS

In our many years of experience in teaching introductory programming we have found this approach to be useful. It gives the students structure and confidence when the program and helps them to focus on the local problems instead of focusing all over the program text. We believe that this way of structuring the course helps especially the weaker students.

The structuring and the content of the course of cause depend on the learning goals for the course. One of our learning goals is that the programming process should be demystified. We have demystified the programming process by focusing on systematic way to convert specifications to working code, thereby postponing the "design" element of the course – the students do not design but are given the design by the lecturer. We believe that a good "reading" ability is a prerequisite for a "writing" ability – in other words the students need to read a lot of contracts and have a good understanding of how one can implement the contract before the students create contracts themselves.

6. REFERENCES

Alphonce, C., and Ventura, P.J.: "Object-Orientation in CS1-CS2 by Design", Proceedings of Innovation and Technology in Computer Science Education, Aarhus, Denmark, 2002.

Arnow, D., Dexter, S., and Weiss, G., Introduction to Programming Using Java: An Object-Oriented Approach, Addison-Wesley, 2004.

Barnes, D.J., and Kölling, M. Objects First with Java – A Practical Introduction using BlueJ, Pearson Education, 2003.

Beck, K., Extreme Programming Explained, Addison-Wesley, 2000.

Bell, D. and Scott, P. 1987. A first course in programming. *SIGCSE Bull.* Vol 19(2) (Jun. 1987). pp. 48-50

Bergin, J., "14 Pedagogical Patterns". Last accessed January 10, 2007. Available on-line at http://csis.pace.edu/~bergin/PedPat1.3.html.

Caspersen, M. E. and Kölling, M. 2006. A novice's process of object-oriented programming. *Companion To the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, October 22 - 26, 2006.

Christensen, H.B., and Caspersen, M.E.: "Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS1", Proceedings of the Fourth Australasian Computing Education Conference, ACE 2000 Melbourne, Australia, 2000.

Cooper, M. et al.: "Teaching Objects-First in Introductory Computer Science", Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, Reno, Nevada, USA, 2003, pp. 191–195.

du Bouley, B (1989). Some difficulties of learning to program. In E. Soloway & J.C. Spohrer (Eds) Studying the novice programmer. Hillsdale, NJ: Lawrence Erlbaum.

Evans, H. and Patterson, W. 1985. Implementing Ada as the primary programming language. In *Proceedings of the Sixteenth SIGCSE Technical Symposium on Computer Science Education* (New Orleans, Louisiana, United States, March 14 - 15, 1985). pp. 255-265

Fjuk, A., Berge, O., Bennedsen, J., & Caspersen, M. E. (2004). Learning object-orientation through ICT-mediated apprenticeship. *ICALT '04: Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT'04),* Joensuu, Finnland. 380-384.

Fleury, A. E. (2000). Programming in java: Student-constructed rules. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 197-201.

Fowler, M., UML Distilled – A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 2000.

Henze, N., Nejdl, W., and Wolpers, M. 1999. Modeling constructivist teaching functionality and structure in the KBS Hyperbook System. In *Proceedings of the 1999 Conference on*

Computer Support For Collaborative Learning. Palo Alto, California, December 12 - 15, 1999.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education), 29*(1), 131-134.

Horstmann, C.S., Big Java, John Wiley & Sons, 2001.

Hyland, E. and Clynch, G. 2002. Initial experiences gained and initiatives employed in the teaching of Java programming in the Institute of Technology Tallaght. Proceedings of the inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on intermediate Representation Engineering For Virtual Machines, 2002 (Dublin, Ireland, June 13 - 14, 2002). pp. 101-106.

Jones, R., Boyle, T. & Pickard, P. (2003) Objectworld: Helping Novice Programmers to Succed through a Graphical Objects-first Approach. Proceedings of 4th Annual LTSN-ICS Conference, NUI Galway, pp. 111 – 114.

Knudsen, J. L., & Madsen, O. L. (1996). Using object-orientation as a common basis for system development education. *ACM SIGPLAN Notices, 31*(12), 52-62.

Knudsen, J.L., and Madsen, O.L. (1998). Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages, proceedings of ECOOP '88 (LNCS 322), p. 21-40. Springer Verlag.

Koffman, E. and Wolz, U. 1999. CS1 using Java language features gently. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on innovation and Technology in Computer Science Education* (Cracow, Poland, June 27 - 30, 1999). pp. 40-43.

Kolb, David A. 1984. *Experiential Learning: Experience as the Source of Learning and Development*. Prentice-Hall, Inc., Englewood Cliffs, N.J.

Kölling, M. & Rosenberg, M. (2001) Guidelines for teaching object orientation with Java, ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education, pp 33 – 36.

Kölling, M: Using BlueJ to Introduce Programming, Chapter 8, this volume.

Kölling, M., Quig, B., Patterson, A. & Rosenberg, J (2003): *The BlueJ system and its pedagogy* Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology, Vol 13, No 4, Dec 2003.

Lewis, J. (2000) Myths about object-orientation and its pedagogy, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, pp. 245-249.

Madsen, O.L., Møller-Petersen, B., and Nygaard, K., Object-Oriented Programming in the BETA Programming Language, Addison-Wesley/ACM Press, 1993.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. “A multinational, multiinstitutional study of assessment of programming skills of first-year CS students”, ACM SIGCSE Bulletin, 33 (4), 2001, pp. 125–140.

Meyer, B. (1992). Applying ‘Design by Contract’. *IEEE Computer*, Vol. 25 (10), October 1992, pp. 40-51.

Nielsen, K. and Kvale, S. (1997). "Current issues of apprenticeship.". *Nordisk Pedagogik* 17 pp. 130 - 139.

Niño J., and Hosch, F.A., (2001). An Introduction to Programming and Object-Oriented Design Using Java, John Wiley & Sons.

Oldham, J. D. 2005. What happens after Python in CS1?. *J. Comput. Small Coll.* 20, 6 (Jun. 2005), 7-13.

Pattis, R. (1993). The ‘Procedures Early’ Approach in CS 1: A Heresy”, Proceedings of the twenty-fourth SIGCSE Technical Symposium on Computer Science Education, pp. 122-126.

Rumbaugh, J., Jacobson, I., Booch, G. (2005) Unified Modeling Language Reference Manual, The, 2nd Edition. Addison-Wesley

Robins, A., Rountree, J. and Rountree N (2003). *Learning and Teaching Programming: A Review and Discussion*, Computer Science Education, Vol. 13, No 2 pp 137 - 172

Schmolitzky, A. (2004) Objects first, interfaces next" or interfaces before inheritance, Educators symposium, OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications pp. 64 – 67.

Shaffer, D. 1986. The use of Logo in an introductory computer science course. *SIGCSE Bull.* Vol 18(4) (Dec. 1986). pp. 28-31.

Sicilia, M. (2006). Strategies for teaching object-oriented concepts with java. *Journal of Computer Science Education, 16*(1), 1-18.

The Joint Task Force on Computing Curricula (IEEE Computer Society and Association for Computing Machinery). Computing Curricula 2001 (final report), December 2001. Available on-line at "http://www.computer.org/education/cc2001/final".

20 Killer “Killer Examples” for Design Patterns

The paper *Killer “Killer Examples” for Design Patterns* presented in this chapter has been published as a conference paper [Alphonce et al. 2007].

[Alphonce et al. 2007] Alphonce, C., Caspersen, M.E. and Decker, A., “Killer ‘Killer Examples’ for Design Patterns”, *SIGCSE '07: Proceedings of the 38th Technical Symposium on Computer Science Education*, Covington, Kentucky, USA, 2007.

Killer "Killer Examples" for Design Patterns

Carl Alphonce
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
alphonce@cse.buffalo.edu

Michael Caspersen
Department of Computer Science
University of Aarhus
DK-8200 Aarhus N, DK
mec@daimi.au.dk

Adrienne Decker
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
adrienne@cse.buffalo.edu

ABSTRACT

Giving students an appreciation of the benefits of using design patterns and an ability to use them effectively in developing code presents several interesting pedagogical challenges. This paper discusses pedagogical lessons learned at the *"Killer Examples" for Design Patterns and Objects First* series of workshops held at the Object Oriented Programming, Systems, Languages and Applications (OOPSLA) conference over the past four years. It also showcases three "killer examples" which can be used to support the teaching of design patterns.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Object-orientation, Design Patterns

1. WHY TEACH DESIGN PATTERNS?

The underlying premise of this paper, and indeed of the workshops from which it derives, is that students need to learn skills and concepts which will be of long-term value to them even as the technology of the day changes. We believe that design patterns are an important part of a student's education in this regard.

However, giving students an appreciation of the benefits of using design patterns as well as an ability to use them effectively in developing code presents several interesting pedagogical challenges. This is especially true for instructors of introductory courses.

The first challenge is that students tend to focus on the input-output behavior of their programs rather than high-level properties of their code. While the input-output behavior of a program is measure of its correctness, there are other aspects of software design that are important. These include the ability of a software solution to scale from small problems to large problems, the degree to which the software is extensible, how robust the software is, and so forth. Workshop participants have observed that students do not pay enough attention to these properties of software in their coursework. Knowledge and use of design patterns highlight these broader issues because they are in large measure the *raison d'etre* of design patterns.

The second challenge to educators is that students often do not believe that what they consider to be very "abstract" design pattern solutions are used or even desirable in fast-paced real-world settings.

A third challenge is that examples which benefit from the application of patterns tend to be more complex (in the sense of involving more code and a "richer" domain) than typical textbook examples. This can make it more difficult for students to grasp the examples and discern their essential characteristics. Pattern catalogs, such as the classic "Gang of Four" [4], are great resources for faculty, but not necessarily for design pattern novices. Faculty need more accessible examples to support their teaching.

A final challenge in teaching design patterns is that examples which are constructed by faculty to demonstrate the power of patterns run the risk of lacking "street cred": students can easily perceive them to be ivory tower products with no grounding in real-world software development. For this reason real-world examples are valuable, since they drive home points better, but they are generally much too complex to present directly to students.

Erich Gamma, in an interview with Bill Venners [5], says of learning design patterns that "You have to feel the pain of a design which has some problem. I guess you only appreciate a pattern once you have felt this design pain." His point is that students will not appreciate design patterns if the are presented to them without an appreciation of the problem they solve. On reflection this makes pretty good sense since a pattern is a solution to a problem in a context. If you don't believe in the problem – if you don't own it– how can you ever appreciate a solution? No problem, no solution!

The "Killer Examples" workshop series was born of a desire to gather examples of design pattern use which address these challenges.

2. WHAT IS A "KILLER EXAMPLE"?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'07
Copyright 2007 ACM .

We define a "Killer Example" to be one which gives overwhelmingly compelling motivation for something. The term is inspired by "Killer App", which is described by the *Jargon File* [1] as, "The application that actually makes a sustaining market for a promising but under-utilized technology."

The "Killer Examples for Design Patterns and Object First" workshops have been held at the OOPSLA (Object-Oriented Programming, Systems, Languages and Applications) conference annually since 2002. In the first four years we have had eighteen examples presented at the workshops. Approaches to teaching design patterns in various settings have also been discussed at these workshops.

This paper shares some of the general lessons we have learned from the workshops, as well as three examples presented at the workshops which we feel best demonstrate what it means to be a "Killer Example".

3. LESSONS LEARNED: THE PEDAGOGY OF "KILLER EXAMPLES"

Many lessons have emerged from the workshops. The most important and recurring ones are described below.

Context Design patterns cannot effectively be taught independent of an application of it. Patterns must be presented in a context which clearly demonstrates the usefulness of the pattern in comparison to the software built without the pattern.

Accessibility Design patterns cannot effectively be taught if the examples used to demonstrate the benefits of the patterns is too complex or too far removed from the experience of students to be meaningful to them.

Real-world Design patterns cannot effectively be taught unless the examples which demonstrate their application and benefits have a real-world grounding. Since patterns are mined from practitioner code, this is important.

Clear benefits Design patterns cannot effectively be taught unless their benefits in terms of desirable high-level properties of software, such as scalability, robustness, extensibility, flexibility and maintainability are clearly evident.

3.1 Intra-Pattern considerations

Although a single "killer example" may demonstrate the use of several design patterns, it is important that for each pattern students move through a sequence of stages of exposure to a single pattern – we therefore refer to these as *intra-pattern* considerations. These stages are motivated by the "read-before-write" pedagogical pattern.[2]

Use it Students should gain an appreciation of the usefulness of a pattern by using an implementation of it. For example, when learning the Iterator pattern students should gain experience by using an Iterator to traverse some collection.

Conceptualize it Students should be engaged in a discussion of the general architecture of a given pattern. For example, when learning the Iterator pattern students must come to understand the concept of an iterator; alternate approaches, such as a cursor, must be discussed.

Build it The next gain in understanding comes from a student's implementation of a pattern. When learning the Iterator pattern students must next create a class that is an iterator over some collection.

Analyze/study high quality code A deeper understanding of any pattern comes from studying a variety of high quality implementations of the pattern. In the case of the Iterator pattern it is perhaps at this point that students begin to truly grasp the beauty of having a separate iterator which can access private parts of a collection; in Java this is achieved by defining a class's iterator as a public inner class.

3.2 Inter-Pattern considerations

At some point the focus must shift from a single pattern back to a system of mutually supporting patterns, as demonstrated in a killer example. At this *inter-pattern* level of experience, we find the following stages:

Design and construct Students must at some point apply their knowledge of patterns to design and construct software. Killer examples can serve as useful exercises for students also in this regard.

Evaluate A final step in the process of learning to use patterns comes in being able to evaluate and critique the use (or lack of use) of design patterns in software.

4. FIRST EXAMPLE: FRAMEWORKS

Software reuse, after decades of unfulfilled promises, is beginning to become true in the form of object-oriented frameworks.[1] Industrial developers can build large, complex software systems that are reliable and computational efficient because they do not build from scratch; the reuse the vast effort invested into software frameworks such as the Java 2 Enterprise Edition, Java Swing, or Remote Method Invocation (RMI).

4.1 Why Frameworks?

Good object-oriented frameworks are unique examples of the strength of the object-oriented paradigm. Looking behind the scenes of good frameworks shows how careful modeling of domain concepts, use of polymorphism, and the use of design patterns makes a piece of software highly flexible and demonstrates the power of low coupling and high cohesion. It is simply a brilliant case study to learn from, and as such the ultimate killer example of the use of design patterns. The framework we present is developed specifically for educational purposes at the introductory level; the framework encapsulates the MVC design pattern.[3]

4.2 Framework Essentials

The essential characteristics of software frameworks are inversion of control and hotspots.

Inversion of control Typical novice programs consist of a number of interacting objects and a single driver that does the setup and defines the main flow of control. The novice programmer applies services provided through classes that are part of the program or through

[1]This example is due to Michael Caspersen. It was presented at the 2003 workshop.

library classes (e.g. collection classes). When programming using a framework, the main flow of control is out of the programmer's sight; it is dictated and controlled by the framework. The novice programmer's task is to supply code that implements interfaces or specializes (abstract) super classes. This is also known as the Hollywood principle: Don't call us, we'll call you.

Hotspots Frameworks define core functionality, control flow, and object collaboration patterns. Application programmers refine frameworks to specific domains by adding code at well-defined points: the hotspots (also known as hooks or variability points). Hotspots can be realized in a number of different ways: call-back methods, delegation to objects implementing interfaces defined by the framework, or subclassing.

A killer example framework must demonstrate the essential characteristics in a simple and convincing way; it must be simple for novices to use and it must be flexible, i.e. allowing a number of distinct, sensible, and interesting instantiations.

4.3 Example: Presenter Framework

The killer example we have chosen is a presenter framework. The presenter framework facilitates construction of multi-media presentations of a domain where the compass-directions are a suitable metaphor for user navigation; Figure 1 demonstrates an instantiation of the framework that shows a presentation of the tomb of Tutankhamon. Using the compass-directions it is possible to visit the different parts of the tomb while pictures and text is being presented to the user.

The presenter framework provides the application programmer with the simple interface shown in Figure 2. This is the hotspot of the framework; an abstract class which the application programmer must specialize to a specific application.

The presenter framework provides the backbone functionality: a large area for displaying images, a smaller one for displaying text, and the four buttons labeled North, East, South, and West. The buttons respond to user clicks by invoking one of the four abstract methods in the abstract class Presenter which the application must specialize.

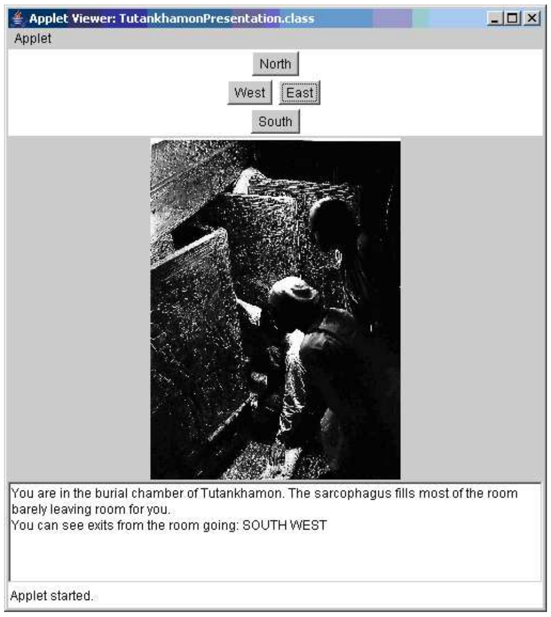
Instantiating the framework is a matter of redefining the four abstract methods in class Presenter (see Figure 3).

4.4 Model-View-Controller in Action

The presenter framework encapsulates the MVC design pattern by defining a View and an abstract Controller which can be plugged with a concrete Controller and a Model to provide a full application. The overall architecture is sketched in Figure 4.

4.5 Discussion

Frameworks can serve in teaching in several ways. At the introductory level frameworks may serve as a black box that makes even a small student effort into a rather impressive program. Later, the black box can be opened to demonstrate how good frameworks are structured. The presenter framework is also used as a stepping stone toward learning more advanced frameworks; the simplicity of the presenter frameworks makes it easier to grasp and understand the



You are in the burial chamber of Tutankhamon. The sarcophagus fills most of the room barely leaving room for you.
You can see exits from the room going: SOUTH WEST

Figure 1: Instantiation of the Presenter Framework

```
public abstract class Presenter {
    public void showImage(String filename) { ... }
    public void showText(String text) { ... }

    public abstract void northButtonPressed();
    public abstract void eastButtonPressed();
    public abstract void southButtonPressed();
    public abstract void westButtonPressed();
}
```

Figure 2: The abstract class Presenter

```
public class TutankahmonPresenter {
    public abstract void northButtonPressed() {
        guest.move(NORTH);
    }
    ....
}
```

Figure 3: Specialization of the abstract class Presenter

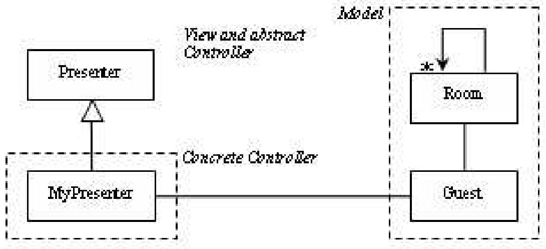


Figure 4: Software architecture

essential characteristics of frameworks (inversion of control and hotspots) and provides a solid ground for working with more complex frameworks (e.g. the Java GUI framework Swing).

We claimed that a killer example framework should allow a number of distinct, sensible, and interesting instantiations. Here are a few such instantiations for this framework:

Virtual museum tour An application which collects pictures of paintings and other artifacts from the Internet and presents a user with a virtual tour of a museum.

Presentation tool The framework forms the core of a presentation tool, along the lines of PowerPoint.

Map navigation Rather than having user interaction generating buttonPressed events, one can have them generated indirectly from a GPS receiver, such that if the coordinates change sufficiently much in a given direction, a buttonPressed event is generated.

While this example does not come directly from a real-world application, the connection to real-world applications is clear and therefore compelling to students.

5. SECOND EXAMPLE: HARDWARE AND SOFTWARE TESTING

Since Design Patterns have grown from the OO community, there are many outside of that community that have difficulty accepting design patterns as applicable to other domains.[2] This is especially true once you leave the software domain and travel to the lands of hardware development, embedded systems, or distributed real-time systems.

In the software domain, and when students study software engineering, an often discussed topic is the idea that when developing large software systems, their development is broken into modules. Those modules are often developed concurrently. The different modules often need to communicate with one another, but development of one module can not stop to wait for another module to be completed.

The same is true in the hardware domain, except some of the modules are software pieces while others are hardware components and their drivers. Developing the software after the hardware is available is often impossible, and both pieces need to be developed and tested concurrently. However, without the hardware to use in the tests, test-driven development, which has shown to be a useful development methodology, can be a challenge.

This example is an industrial example that has been used by a company that develops real-time and embedded systems. They needed to devise a way to develop and test their entire product, the software components and the hardware it will run on concurrently.

5.1 A First Attempt

A naïve attempt to solve this problem is shown in figure 5. A test case is written (TestCase) to test the class/component (ClassUnderTest). ClassUnderTest requires one or more of the hardware components controlled by drivers A, B, and C. The problem with this design stems from the fact that the hardware components are still under development and therefore their drivers are not available. Therefore, testing the class would not be possible until the drivers become available. However, we can introduce a solution which allows us to program to an interface, not an implementation and complete the testing of ClassUnderTest.

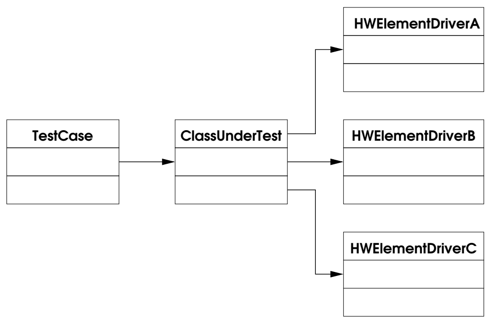


Figure 5: Naïve implementation

5.2 The Strategy Pattern

If we introduce the strategy pattern to this problem, we create interfaces for each one of the hardware driver elements. The drivers themselves have yet to be written because the hardware components are not yet completed. The introduction of interfaces enforces what the drivers will look like (i.e. what methods the drivers will contain). Then, testing can be completed of ClassUnderTest before the hardware is ready. Also, when introducing this pattern, we allow for differences in the underlying implementation of the drivers (i.e. multiple classes that implement the driver interface but actually connect to different hardware implementations).

5.3 The Abstract Factory Pattern

Introducing the Strategy pattern allows us to test the ClassUnderTest independent of the hardware or hardware driver implementations. However, we could have introduced a potential problem. Suppose that some implementations of the driver for hardware component A, only work with certain other configurations of hardware components B and C. We need a way to ensure that the correct configuration of hardware components and drivers are tested. Thus, the introduction of an abstract factory becomes necessary to manage the configurations of the drivers for the hardware components. Then, the TestCase can interact directly with the factory to invoke the proper configuration of the hardware when testing the ClassUnderTest.

Applying both these patterns results in a design as shown in figure 6.

5.4 Why this example is Killer

The applications of the patterns to help solve this problem are not buried in the complexity of the solution to understand. It illustrates the fact that patterns do not always exist in isolation and the introduction of one pattern often necessitates the introduction of more. This example also illustrates that design patterns are not limited to organizations that strictly develop software, but can be used to work with embedded and real-time systems development. It also shows how design patterns can support the test-driven development methodology.

[2]This example is due to Bruce Trask and Angel Roman. It was presented at the 2005 workshop.

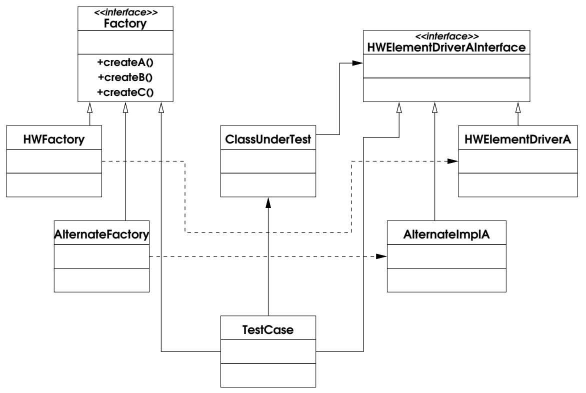


Figure 6: A more flexible approach

6. THIRD EXAMPLE: INTERACTIVE PROGRAM GUIDE

An interactive program guide (IPG) allows a user to browse television (cable/satellite) content in various ways, such as by channel, title, timeslot, and genre.[3] Some systems provide access to weather forecasts. It is also possible to use the IPG to set subtitle or closed captioning options. To control the IPG a user presses keys on a remote control. The remote control typically has a small number of buttons used for navigation and selection. Depending on the current state of the IPG system, different things might happen when a given button is pressed.

For example, selecting a program to watch in the normal TV mode will switch to the indicated channel. However, in pay-per-view (PPV) mode some additional level of confirmation is required, so that a user does not accidentally incur a charge for a program they do not wish to pay for.

Similar systems are used in hotels to present guests with various kinds of information. For example, hotel systems allow guests to order things as diverse as movies and room service. They typically also allow guests to view their hotel bill on-screen and also to check out.

This example is especially interesting because it is a real-world example combining a large number of patterns which nonetheless is accessible. Among the many patterns incorporated in this example are state, model-view-controller, observer, iterator, composite, command, singleton, and proxy. The role of a few of these patterns in the example is presented below.

6.1 Iterator Pattern

The iterator pattern is used to allow the IPG system to traverse a variety of data structures, representing things such as channels, groups of channels, programs, etc. The IPG system maintains a "current" position during browsing, something that lends itself to implementation using a bi-directional iterator.

6.2 State Pattern

An obvious design issue is that the system is *state-based*. In other words, its behavior is governed by the particular state that it is in. Indeed, the behavior associated with all the buttons on the controller change together as the state of the IPG changes. This is modelled this using a state pattern.

Using the state pattern in this example helps to ensure robustness: the behavior of the system is always coherent, since the behaviors associated with a collection of buttons is changed *en masse*.

[3]This example is due to Asher Sterkin. It was presented at the 2003 workshop.

6.3 Command Pattern

The command pattern is used to represent the behaviors associated with particular buttons on the controller. Because these behaviors are "objectified" as command objects the system retains the flexibility to easily accomodate new menus with new features.

6.4 Mediator Pattern

The mediator pattern is used to maintain loose coupling between components in the case where the IPG displays category information in one pane and element information in another, and changes to the category must result in changes to the set of elements displayed.

6.5 Discussion

This example has demonstrated the potential application of a handful of design patterns in a real-world software system. The beauty of this particular example is that it is one that is familiar to most, if not all, students. The domain of the problem is therefore immediately accessible to them.

7. CONCLUSION

In this paper, we have discussed three "Killer Examples" that introduce students to problems that lend themselves nicely to solutions using design patterns. Many of the complaints of instructors about teaching design patterns stem from the inability to find examples that show the utility of patterns. Many examples are of "toy problems" that do not show the usefulness of the pattern in a larger context, or the examples involve a system that is too complex to break down. A unique balance has been reached in these three examples that allows an instructor to provide a problem and a problem domain that is accessible to students that points to where design patterns can be useful and beneficial to the overall system.

The example used to illustrate patterns is arguably the "make or break" point in a student's pattern education. If patterns are presented as some lofty educational-only idea, students will not see them for their usefulness in real-world software development settings. If patterns are viewed and presented by educators as a real-world-only problem, then students will miss out on an opportunity to be exposed to a beneficial tool for software engineering early in their careers.

8. REFERENCES

[1] The jargon file. http://catb.org/~esr/jargon/.

[2] J. Bergin. Some pedagogical patterns. http://csis.pace.edu/ bergin/patterns/fewpedpats.html.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] B. Venners. How to use design patterns – a conversation with Erich Gamma, part I. 2005.

21 Assessing Process and Product

The paper *Assessing Process and Product — A Practical Lab Exam for an Introductory Programming Course* presented in this chapter has been published as a conference paper [Bennedsen et al. 2006b].

[Bennedsen et al. 2006b] Bennedsen, J. and Caspersen, M., "Assessing process and product — A practical lab exam for an introductory programming course", *Proceedings of the 36th Annual Frontiers in Education Conference,* San Diego, California, pp. M4E-16-M4E-21, 2006.

Assessing Process and Product
– A Practical Lab Exam for an Introductory Programming Course

Jens Bennedsen[1] and Michael E. Caspersen[2]

Abstract - The final assessment of a course must reflect its goals, and contents. An important goal of our introductory programming course is that the students learn a systematic approach for the development of computer programs. Having the programming process as learning objective naturally raises the question how to include this in assessments. Traditional assessments (e.g. oral, written, or multiple choice) are unsuitable to test the programming process.

We describe and evaluate a practical lab examination that assesses the students' programming process as well as the developed programs. The evaluation is performed in two ways: By analyzing the results of two lab examinations (with more than 500 students) and by semi-structured individual interviews with representatives of the involved persons (students, TAs, lecturer, and examiner).

The result of the evaluation is encouraging and indicates the value of alignment and strong conformity between goal, content and assessment of the introductory programming course.

Index Terms – CS1, Examination, Evaluation, Programming Process, Objects-First, Pedagogy.

INTRODUCTION

The final assessment must reflect aims, goals, and contents of a course [1].

An important goal of our introductory programming course is that the students learn a systematic approach to the development of computer programs. Learning a systematic approach to programming implies that the students must gain a clear understanding of the programming process and the activities that are part of this process. They must also develop the ability to apply these to develop programs.

Recognizing the importance of programming techniques and the programming process when designing a programming course implies the need for adoption of a suitable assessment form. Traditional assessment forms (e.g. oral or written examinations, multiple choice questions) are unsuitable to test the programming process.

Another equally important argument for assessing the programming process is that "The spirit and style of student assessment defines de facto the curriculum" [2][p.1]. Ramsden makes a similar observation: "the type of grading influences the student's learning approach" [3].

The bottom line is that it is essential to apply an evaluation form where the students demonstrate their practical programming skills as well as their understanding of the fundamental concepts and theories from the curriculum of the course. Consequently, we need to develop a new type of assessment suitable to test the programming process as well as the product.

The lab examination described and evaluated in this paper has as characteristics that it

i. provides a valid and accurate evaluation of the student's programming capabilities,
ii. evaluates the process as well as the product,
iii. encourages the students to practice programming throughout the course, and
iv. can be used assess 120-140 students pr. day.

The rest of the paper is structured as follows: Section 2 describes the context of the lab examination. Section 3 gives a more thorough description of the final lab examination. Section 4 presents and discusses the findings from the evaluation of the lab examination. In section 5 we discuss related and future work. The conclusions are drawn in section 6.

GOALS, CONTENT AND ASSESMENT

To provide an understanding of the context, this section describes goal, form, and content of the introductory programming course.

General Information

Our programming course spans the first half of CS1 at University of Aarhus. The course runs for seven weeks, and after the course there is a lab examination with a binary pass/fail grading.

The grading is based solely upon the behaviour in and result of the final examination; acceptable performance during the course is a prerequisite for the final exam but does not count as part of the grading.

There are approximately 250 students per year from a variety of study programmes, e.g. computer science, mathematics, geology, nano science, economy, multimedia. 40% of the students are majors in computer science, and they are the only group of students that continue with the second half of CS1. The rest of the students proceed to other programming courses related to their fields (e.g. multimedia

[1] Jens Benendsen, IT University West, Fuglsangs Alle 20, DK- 8210 Aarhus V, Denmark, jbb@it-vest.dk
[2] Michael E. Caspersen, Department of Computer Science, University of Aarhus, DK-8200 Aarhus N, Denmark, mec@daimi.au.dk

programming, scientific computing) if they proceed with programming at all.

The students are grouped in teams of 18-20 students; typically there are 13-14 teams per year. Each team has its own teaching assistant (TA) – a PhD or MSc student.

Goals

The purpose of the course is that students learn the foundation of systematic construction of simple programs and through this obtain knowledge about the role of conceptual modelling in object-oriented programming. Furthermore, it is the goal that students become familiar with a modern programming language, fundamental programming language concepts, and selected class libraries.

After the course the students must be able to explain and use fundamental elements in a modern programming language, use conceptual modelling in relation to preparing simple object-oriented programs, implement simple object-oriented models in a modern programming language, and use selected class libraries.

Form

The course runs for seven weeks; every week there are four lecture hours and one lab hour plus three class hours with a TA. In addition to the scheduled hours, students work approximately seven hours per week in study groups or on their own.

The four lecture hours per week are used for presentation and discussion of general concepts and the programming process. The programming process is revealed through live programming in front of the students in the lecture theatre using computer and projector and through process recordings (narrated, screen-captured video recordings of program development sessions), see [4].

Every week (except for the first) there is a mandatory assignment that must be submitted to the TA. The TA examines the assignments and gives personal as well as collective feedback to the students. Approval of five out of six weekly assignments is a prerequisite for the final exam but does not count as part of the grading. The weekly assignments are primarily used to keep the students up to the mark on the practice of programming.

Content

The course content is fundamental programming language concepts, object-orientation, and techniques for systematic construction of simple programs.

- **Fundamental programming language concepts**: variable, value, type, expression, object, class, encapsulation, control structure, method/procedure, recursion, type hierarchies.
- **Object-orientation**: modelling; class structures (specialization, aggregation and association); use of selected class libraries (in particular collection libraries), interfaces and abstract classes.
- **Systematic development of small programs**: modularization, stepwise refinement/incremental development, test.

This is a logical listing of the course contents; it is *not* the order in which the content is covered. The content is covered using a spiral approach [5]; for further details of the structure and content of the course, see [6, 7].

Assessment Through a Lab Examination

This section discusses the examination requirements, the organization of the lab examination and the actual lab examination.

Conformity between Goals, Content, and Assessment

As mentioned in section “Goals”, the goals of the course are that the student must be able to explain and

- *use* fundamental elements in a modern programming language,
- *use* conceptual modelling in relation to preparing simple object-oriented programs,
- *implement* simple object-oriented models in a modern programming language, and
- *use* selected class libraries.

During the course, as in real life, programs are developed using a standard development environment running on a computer. An ordinary written exam with pen and paper is an artificial situation and therefore insufficient and inappropriate to test the student’s ability to develop programs. For the same reasons an ordinary oral examination and a multiple choice test would be inappropriate.

To ensure alignment and maximum conformity between goals, content, and assessment we have designed a practical examination organized in a lab.

Organization of the Lab Examination

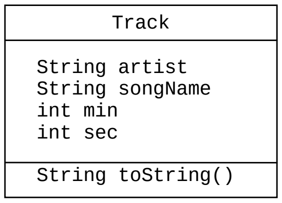
The examination resembles an ordinary lab session. 20 students are tested concurrently.

We schedule one hour per group of 20 students, but only 30 minutes for the actual lab examination. The rest of the time is used for administrative activities and as buffer.

Each group of students receives a different assignment consisting of nine small progressive programming tasks. In principle the assignments are identical (they are all instances of the same generic assignment), but the students does not know nor realize this. The similarity of the assignments is important for fairness as well as comparability of the students’ results. The sample assignment in Figure 1 deals with tracks and play lists; other exercises concern luggage and flights, employees and departments, museums and paintings, etc. Although the concepts modelled by the classes vary, the assignments have similar structure.

Lab Exam Exercise (30-minute exam)

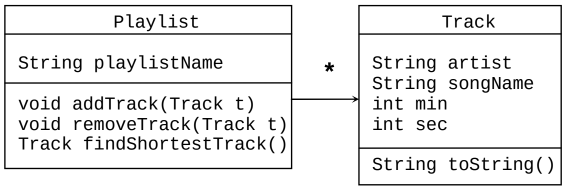
1. Create a class, *Track*, that represents a piece of music; the *Track* class is specified in the following UML diagram.



The four field variables must be initialized in a constructor (through four parameters of suitable types). The method *toString* must return a string representation for a piece of music, e.g.

"Yesterday: The Beatles (2:05)"

2. Create a test method named *exam* in class *Driver*. The method must be static, have return type void, and have no parameters.
3. Create two *Track* objects in the exam method using object references *t1* and *t2*; print the two *Track* objects using the *toString* method.
4. Create a new class, *Playlist*, representing a collection of *Tracks*; the *Playlist* class and its relation to the *Track* class is specified in the following UML diagram:



5. Implement the method *addTrack* (and *removeTrack*) so that it adds (removes) the object *t* to (from) the *Playlist* object.
6. Create a *Playlist* object in the *exam* method in the *Driver* class; associate the two existing *Track* objects with the *Playlist* object.
7. Implement the method *findShortestTrack*. The method must return a shortest (measured in playing time) *Track* object from a *Playlist* object. You can assume a non-empty *Playlist* object. In other words, you need not worry about the playlist being empty.
8. Use methods *findShortestTrack* (from class *Playlist*) and *toString* (from class *Track*) to print the shorter of the two *Track* objects created in task 3.
9. Let the *Track* class implement the *Comparable* interface. The natural order of *Track* objects is defined by the length of the song.

Figure 1: *Sample Lab Exam Exercise*

Assessment of Product and Process

In test for completed apprenticeship of traditional crafts, the examiner inspects the apprentice while they construct their exam product; the quality of the apprentice's construction process as well as the quality of the final product counts in the final grading.

Because of similar goals regarding the assessment of process and product, we have adopted a similar examination form where the lecturer and the external examiner evaluate the programming process as well as the program produced by each student by inspecting the students during the examination.

To avoid practical problems during start-up and finalization of the lab examination (e.g. login problems, applying naming conventions, delivery of the exam products), and to ensure that minor unimportant programming errors, tool problems, etc. does not hinder the student's problem solving and programming, five TAs are present during the lab examination to support the students. If the TAs have doubts about their role (e.g. how much to interact with the students), they consult the lecturer or external examiner on-the-fly.

To let the students settle down and get started, they are not inspected until they have passed a checkpoint after the first three programming tasks. The students are instructed to call upon a TA or the lecturer when they reach the checkpoint to show and demonstrate their solution. When a student has passed the checkpoint, the lecturer and external examiner start inspecting the student's behaviour. The poorest students never reach the checkpoint i.e. the inspection time is focused on those students who have a chance of passing.

The examiner and lecturer note the time when the first three tasks are done. After five to seven minutes, they start inspecting the process of each student; around that time, and after a short inspection of the students programming process, it is usually possible to determine the pass grade. This is a very efficient way to know when and in what order to look at the students' solutions. This is also a method to ensure that the students have some silence and can concentrate during the exam.

To allow for efficient inspection, the students are instructed to keep all editor windows open and tiled on the screen.

The students' behaviour as well as the quality of the programs they produce count in the final grading but not on equal footing. An appropriate and systematic programming process can compensate for minor flaws and errors in the product and result in a pass mark for the student, and similarly a poor process can be the determining factor when the product is on the edge. Although we emphasize the programming process, it is not the case that a nice product will be turned down due to a poor process (which is unlikely anyway).

EVALUATION

In this section, we present and discuss an evaluation of the lab examination described above.

Evaluation Method

The evaluation of the lab exam was performed in two ways: By analyzing the results of three consecutive lab examinations (2003, 2004 and 2005) and by semi-structured individual interviews with students, TAs, the examiner, and the lecturer.

Quantitative Evaluation

For each of the three years we have collected data about the students for four variables (and two derived). The description of the variables can be found in Table 1.

Variable	Description
students	students enrolled for the course
abort	students that aborted the course before the final exam
exam	students allowed to take the final exam
skip	students that did not show up for the final exam but was allowed to
fail	students who failed the final exam
pass	students who passed the final exam

Table 1: *Description of type of data*

The numbers in table 1 are related as follows:

students = *abort* + *exam*
exam = *skip* + *fail* + *pass*

From these numbers we calculate *exam rate*, *pass rate* and *retention rate* (*exam*/*students*, *pass*/*exam*, *pass*/*students*). The results are presented in Table 2.

	2003	2004	2005
students	276	220	295
abort	63	26	28
exam	213	194	267
exam rate	77.2 %	88.2 %	90.5 %
skip	13	5	3
fail	15	19	29
pass	185	170	235
pass rate	86.9 %	87.6 %	88.0 %
retention rate	67.0 %	77.3 %	79.7 %

Table 2: *Statistics from three years of practical lab exams*

The figures in Table 2 reveals two interesting aspects: the improved exam rate (and retention rate) from 2003 to the following years, and the high pass rate in general.

The curriculum was radically redesigned in 2003 going from a semester structure to a quarter structure; consequently the traditional CS1 course was split in two courses with an exam in between. The students of 2003 were the first to take the new course with the new examination form, and therefore there where no tradition for the students to lean on. In the following years (2004-2005) the students have had the old exam questions to use for practice, and older students to hear war stories from. In the following years the lecturer could be more explicit when describing the requirements for the exam and the exam form. We believe that this is the primary reason for the improved exam rate.

The pass rate is high compared to what others report [8, 9]. We believe that this primarily is due to the alignment and the strong conformity between goal, content and assessment of the course.

Qualitative Evaluation

The semi-structured interviews were conducted two to three weeks after the final exam. Ten students were selected to get a mixture of major and gender. One interviewer conducted each interview. The interviews were audio taped for later analysis. The interviews followed an interview guide focusing on three topics: The lab exam form in general, this specific exam, and the evaluation form compared to other evaluation forms. In the analysis that follows, quotations from the interviews are presented that describes the general attitude of the group. The interviews were done in Danish, and the quotations translated into English by the authors.

The Students

There was a very little difference in the way that the interviewed students had experienced the lab exam; their answers were largely similar. We find therefore that the students are representative of the general attitude towards the exam, although we cannot be sure.

All of the interviewed students found the evaluation form fair. They defined *fair* as "*if you have practiced during the course, you can expect to pass the exam*". They all found that the form and content of the exercise was very adequate with respect to the goals of the course. As one student noticed: "*Programming requires very abstract thinking, but it is also a craft ... the examination form perfectly suits this mixture.*"

One of the students did not like that a TA was looking over her shoulder. She felt insecure and nervous. However, she was the only one having this experience – no one else minded having the TAs around (some even found their presence to give more peace of mind).

The examination incited the students to practice programming. As an option for the students, exam exercises from the previous year were available for preparation for the exam. As one student replied when asked about his preparations, "*I solved all the [old] exam exercises*".

Students were instructed to call the TA after solving the first three tasks of the exercise (Figure 1) to demonstrate what they had achieved. None of the students found this to be problematic, but some of them pointed to the possible problem, that the slow students might feel this as an extra stress factor (knowing that many of the other students have finished). In conclusion, only one of the interviewed students felt the examination to be stressful.

All of the interviewed students felt that a more fine-grained marking could take place, but it would require more time and more tasks. Most thought that one hour would be sufficient for this.

The Teaching Assistants

The interviews with the teaching assistants in many ways supported the statements from the students. They also found the exam to be fair and had the impression that it evaluates the students programming skills.

In the beginning, the TAs had some difficulties knowing to what extent they could answer questions. During the exam, the TAs developed a practice: they helped a student who had spent several minutes trying to figure out a simple problem, but did not help with problems that were more fundamental. If in doubt, the TAs asked the lecturer or examiner. Apart from this, they did not feel uncomfortable with they role.

The Lecturer and the External Examiner

Both the lecturer and the examiner found the exam form to be both fair and evaluating the learning objectives of the course. The external examiner found that the exam evaluated the student's understanding of the general concepts although it was impossible to evaluate that the student was "*able to explain [...] fundamental elements in a modern programming language*". They found that it was easy to assess an objective pass/fail criterion due to the generic exercises. The examiner thought that a little longer time would give an even better evaluation criterion.

The examination gave a good impression of the students programming skills including their programming process. As the examiner said: "*When you get an error message from the compiler you must be able to figure out what is wrong … that is a part of a practical programming skill*".

Concluding the Evaluation

The exam tests the process as well as the product. In some cases the process was the decisive factor. One special example of this was a student that was ill and therefore worked very slowly; however slow, her programming process was very good demonstrating a systematic approach to solving the problems.

The evaluation indicates that the lab examination supports the learning objective of the course. The students and the lecturer/examiner consider the lab examination fair. The assessment does not require many resources: 250 students can be handled using less than 90 person-hours.

Low retention is one of the main problems in CS1 courses. As noticed by [10][p.40] their retention "has been around 50%". In this course, the retention is around 75%. We have found that the examination form kept the students up to the mark; they did actually practice programming. We think this is one of the explanations of the relatively high retention rate.

For computer science students the examination form must be seen in conjunction with the examination form of the following course (the second part of CS1), which is an oral examination focusing more on the conceptual aspects of introductory programming. There is a progression from the first exam to the next, from testing practice to testing conceptual knowledge.

RELATED AND FUTURE WORK

Recently, a growing number of papers reporting on laboratory exams for introductory programming courses have been published [11-15]. All report good results using this apparently novel assessment form. However, a common characteristic of the assessment methods presented in these articles, and a deficiency compared to the method described herein, is that the evaluation and grading is based solely upon the end product, the students' final solutions.

In [12] the authors describe the grading in their lab final (their word for lab exam): "*Grading on the exam is focused on working programs*". Only the result of the process is evaluated, not the process. Barros [11][p.18] report on the use of lab exams during the course, but the final exam is a traditional written exam. The "*rationale behind maintaining code written in the final exam was to evaluate the students in an environment where trial and error is simply not possible*". Again, they do not include an evaluation of the programming process in their lab exam; the focus is on the final product only.

Focus on the programming process during the course is very important. We are currently investigating the idea of having the students supply information about their programming process (in the form of a screen capture of a programming session) and include this as part of their weekly, mandatory assignment. We expect this information to be valuable and useful for the TAs and the lecturer in order to provide feedback on the process as well as the product, and in general to improve the ability to address the actual needs of the students.

CONCLUSION

We have described and evaluated a lab exam which has a number of advantages. It is simple to evaluate the student's programming process as well as the product (the result of the student's efforts). It is a fair and effective exam. We use standardized exercises that each covers more than 80% of the curriculum. The environment for the exam is the normal daily work environment. It is a lightweight exam easy to prepare and carry out. It requires a couple of days to prepare the exercises for the exam, and we had a throughput of 100 students per day. Everyone involved, in particular the students, regard form as well as content of the exam to be very good and in excellent correspondence with the learning objectives of the course.

ACKNOWLEDGEMENT

It is a pleasure to thank Gudmund Frandsen for valuable comments during development and practice of the lab exam described and evaluated in this paper. We will also like to thank the students and TAs who participated in the interviews. A special thank to Michael Kölling for valuable comments on an earlier version of this article.

References

[1] J. C. Prior and R. Lister, "The backwash effect on SQL skills grading," in *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* 2004, pp. 32-36.

[2] D. Rowntree, *Assessing Students. how Shall we Know them?* , vol. rev. ed., repr., London: Kogan Page, 1988,

[3] P. Ramsden, *Learning to Teach in Higher Education.* London: Routledge, 1992,

[4] J. Bennedsen and M. E. Caspersen, "Revealing the programming process," in *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* 2005, pp. 186-190.

[5] J. Bergin. Fourteen pedagogical patterns. Available: http://csis.pace.edu/~bergin/PedPat1.3.html

[6] J. Bennedsen and M. E. Caspersen, "Programming in context: A model-first approach to CS1," in *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* 2004, pp. 477-481.

[7] M. E. Caspersen and H. B. Christensen, "Here, there and everywhere - on the recurring use of turtle graphics in CS1," in *ACSE '00: Proceedings of the Australasian Conference on Computing Education,* 2000, pp. 34-40.

[8] R. Andersson and T. Roxå , "Encouraging students in large classes," in *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* 2000, pp. 176-179.

[9] J. Börstler, T. Johansson and M. Nordström, "Teaching OO concepts - a case study using CRC-cards and BlueJ," in *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference,* 2002, pp. T2G-1-T2G-6.

[10] A. N. Kumar, "The effect of closed labs in computer science I: an assessment," *J. Comput. Small Coll.,* vol. 18, pp. 40-48, 2003.

[11] J. P. Barros, L. Estevens, R. Dias, R. Pais and E. Soeiro, "Using lab exams to ensure programming practice in an introductory programming course," in *ITiCSE '03: Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education,* 2003, pp. 16-20.

[12] M. E. Califf and M. Goodwin, "Testing skills and knowledge: Introducing a laboratory exam in CS1," in *SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education,* 2002, pp. 217-221.

[13] A. T. Chamillard and K. A. Braun, "Evaluating programming ability in an introductory computer science course," in *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* 2000, pp. 212-216.

[14] C. Daly and J. Waldron, "Assessing the assessment of programming ability," in *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* 2004, pp. 210-213.

[15] N. Jacobson, "Using on-computer exams to ensure beginning students' programming competency," *SIGCSE Bull,* vol. 32, pp. 53-56, 2000.

22 Beauty and the Beast

The paper *Beauty and the Beast — Toward a Measurement Framework for Quality of Example Programs* presented in this chapter has been submitted for ITiCSE 2007 [Börstler et al. 2007].

[Börstler et al. 2007] Börstler, J., Caspersen, M.E. and Nordström, M., "Beauty and the Beast — Toward a Measurement Framework for Quality of Example Programs", submitted for *ITiCSE '07: The 12th international conference on Innovation and Technology in Computer Science Education,* Dundee, Scotland, 2007.

Beauty and the Beast

Toward a Measurement Framework for Quality of Example Programs

Jürgen Börstler
Department of Computing Science
Umeå University
SE-90187 Umeå, Sweden
jubo@cs.umu.se

Michael E. Caspersen
Department of Computer Science
University of Aarhus
DK-8200 Aarhus N, Denmark
mec@daimi.au.dk

Marie Nordström
Department of Computing Science
Umeå University
SE-90187 Umeå, Sweden
marie@cs.umu.se

ABSTRACT

Examples are important tools for programming education. In this paper, we investigate desirable properties of programming examples from a cognitive and a measurement point of view. We argue that some cognitive aspects of example programs are "caught" by common software measures, but they are not sufficient to measure understandability of examples. We conclude that a framework for measuring understandability of examples should also consider factors related to the usage of the example.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education*.

General Terms

Design, Measurement.

Keywords

CS1, Programming Examples, Measurement, Understandability.

1. INTRODUCTION

Example isn't another way to teach. It is the only way to teach. [A. Einstein]

Examples are important teaching tools. Research in cognitive science confirms that "examples appear to play a central role in the early phases of cognitive skill acquisition" [34]. More specifically, research in cognitive load theory has shown that alternation of worked examples and problems increase learning outcome [31].

Students use examples as templates for their own work. Examples must therefore be consistent with the principles and rules of the topics we are teaching and free of any undesirable properties or behaviour. If not, students will have a difficult time recognizing patterns and telling an example's superficial surface properties from those that are structurally important.

Perpetually exposing students to "exemplary" examples, desirable properties are reinforced many times. Students will eventually recognize patterns of "good" design and gain experience in telling desirable from undesirable properties. Trafton and Reiser [32] note that in complex problem spaces, "[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples".

With carefully developed examples, we can minimize the risk of misinterpretations and erroneous conclusions, which otherwise can lead to misconceptions. Once established, misconceptions can hinder students in their learning and be difficult to resolve [8, 27].

But how can we tell "good" from "bad" examples? Can we measure the quality of an example?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE 2007, June 25–27, 2007, Dundee, Scotland.
Copyright 2007 ACM X-XXXXX-XX-X/XX/X.

2. PROPERTIES OF GOOD EXAMPLES

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. [M. Fowler]

Programming is a human activity, often done in teams. About 40-70% of the total software lifecycle costs can be attributed to maintenance and the single most important cost factor of maintenance is program understanding [33]. That said, Fowler makes an important point in the quote above. In an educational context, this statement is even more important. In the beginning of their first programming course, students can't even write a simple program that a computer can understand.

A good example must obviously be *understandable by a computer*. Otherwise it cannot be used on a computer and would therefore be no real programming examples.

A good example must also be *understandable by students*. Otherwise they cannot construct an effective mental model of the programs. Without "understanding", knowledge retrieval works on an example's surface properties only, instead of on its more general underlying structural properties [10, 32, 34].

A good example must also *effectively communicate the concept(s) to be taught*. There should be no doubt about what exactly is exemplified. To minimize cognitive load [25], an example should furthermore only exemplify one (or very few) new concept at a time.

The "goodness" of an example also depends on "external" factors, like the pedagogical approach taken. E.g., when our main learning goal is proficiency in object-oriented programming (in terms of concepts, not specific syntax), our examples should always be truthfully object-oriented and "exemplary", i.e. adhere to accepted design principles and rules and not show any signs of "code smells" [12, 22, 28]. If examples are not always truthfully object-oriented, students will have difficulties picking up the underlying concepts, principles, and rules.

These three properties might seem obvious. However, the recurring discussions about the harmfulness or not of certain common examples show that there is quite some disagreement about the meaning of these properties [37, 1].

3. SOFTWARE MEASUREMENT

When you can measure what you are speaking about, and express it in numbers, you know something about it; but *when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind.* [Lord Kelvin]

From our discussion in the previous sections, it would be useful to find some way of determining the understandability of a programming example. A suitable measure could help us choose between examples and guide the shaping of examples.

According to SEI's quality measures taxonomy, understandability is composed of *complexity*, *simplicity*, *structuredness*, and *readability* [29]. Bansiya and Davis [3] describe understandability as "[t]he properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure".

There are large bodies of literature on software measurement [14, 26, 4] and program comprehension [6, 21, 5, 13]. The work on software measurement focuses mainly on the structural complexity of software. There is only little work on measuring the cognitive aspects of complexity [7, 30]. The work on program comprehension focuses on the cognitive aspects, but is mainly concerned with the comprehension process and not with software measurement.

4. ONE PROBLEM, TWO SOLUTIONS

Technical skill is mastery of complexity, while creativity is mastery of simplicity. [C. Zeeman]

Let us forget for a moment about actual software measures and look at two example programs for implementing a class *Date*: the Beauty and the Beast.

4.1 The Beauty

The Beauty (Figure 4-2) is developed according to sound principles of decomposition; we could call it *extreme decomposition*. The Beauty consists of four classes: *Date* with components *Day*, *Month*, and *Year*. A *Date* object knows its *Day*, *Month*, and *Year*. The three classes *Day*, *Month*, and *Year* are encapsulated as inner classes of the *Date* class, since they are not relevant to the surroundings. Their existence is a result of our choice of representation for class *Date* (see Figure 4-1).

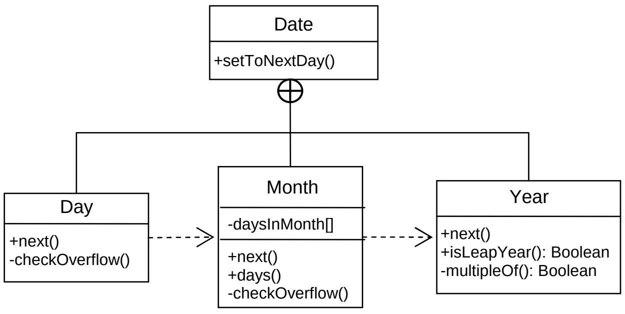


Figure 4-1: *UML diagram for the Beauty*

The Beauty is beautiful for several reasons. First, there is an explicit representation of each of the key concepts in the problem domain. These can work as clues (beacons) aiding in code comprehension [13]. Second, the interfaces and implementations of all classes are very simple and represent an easily recognizable distribution of responsibilities. Third, carefully chosen identifiers, matching problem domain concepts, enhance the readability of the code. Fourth, extreme decomposition supports independent and incremental comprehension, development, and test of each of the four component classes, as well as each of the methods in the classes.

A drawback of The Beauty is that one has to look into several classes to get the full picture of the solution. This problem can, however, be easily solved by providing a class diagram, like the one in Figure 4-1.

```
public class Date_Beauty {
  private Day day;
  private Month month;
  private Year year;

  public Date_Beauty(int y, int m, int d) {
    this.year = new Year(y);
    this.month = new Month(m);
    this.day = new Day(d);
  }
  public void setToNextDay() {
    day.next();
  }

  private class Day {
    private int d;   // 1 <= d <= month.days()

    public Day(int d) {
      this.d = d;
    }
    public void next() {
      d = d + 1;
      checkOverflow();
    }
    private void checkOverflow() {
      if ( d > month.days() ) {
        d = 1;
        month.next();
      }
    }
  } // Day

  private class Month {
    private int m;   // 1 <= m <= 12
    private final int[] daysInMonth=
      {0,31,28,31,30,31,30,31,31,30,31,30,31};
      /*  1  2  3  4  5  6  7  8  9 10 11 12 */

    public Month(int m) {
      this.m = m;
    }
    public int days() {
      int result= daysInMonth[m];
      if ( m == 2 && year.isLeapYear() ) {
        result = result + 1;
      }
      return result;
    }
    public void next() {
      m = m + 1;
      checkOverflow();
    }
    private void checkOverflow() {
      if ( m > 12 ) {
        m = 1;
        year.next();
      }
    }
  } // Month

  private class Year {
    private int y;

    public Year(int y) {
      this.y = y;
    }
    public void next() {
      y = y + 1;
    }
    public boolean isLeapYear() {
      return (isMultipleOf(4) && !isMultipleOf(100))
             || isMultipleOf(400);
    }
    private boolean isMultipleOf(int a) {
      return ( y % a ) == 0;
    }
  } // Year
} // Date_Beauty
```

Figure 4-2: *The Beauty*

4.2 The Beast

The Beast (Figure 4-3) is structured as one monolithic method. We could say it was developed according the principle of *no decomposition*.

The Beast has the advantage of collecting everything in one place. This leads to much less code in total. All necessary information is contained in a single statement sequence. The drawbacks are however numerous.

First, there is no explicit representation of the key concepts in the problem domain. Although this solution is much smaller than The Beauty, it is nevertheless difficult to get the full picture. It is not even possible to provide a high-level diagram to resolve that problem, since all processing is contained in a single method. Second, there is mainly one long statement sequence where everything is happening. Such an approach makes it impossible to introduce meaningful identifiers as clues (beacons) aiding in code comprehension. Third, the Beast shows no signs of "work units" or "chunks" of information. That makes it difficult to deconstruct the program and find appropriate starting points for a code comprehension effort. Students might furthermore conclude that such a program is constructed as a large monolithic unit. The Beast does not lend itself as a pattern for incremental testing and development. Fourth, The Beast is highly nested. Students have to keep track of many conditions at the same time, which increases cognitive load [25].

```
class Date_Beast {
  private int day;   // 1 <= day <= days in month
  private int month; // 1 <= month <= 12
  private int year;

  public Date_Beast(int y, int m, int d) {
    day = d;
    month = m;
    year = y;
  }
  public void setToNextDay() {
    int daysInMonth;
    if ( month == 1 || month == 3 ||
         month == 5 || month == 7 ||
         month == 8 || month == 10 ||
         month == 12 ) {
      daysInMonth = 31;
    } else {
      if ( month == 4 || month == 6 ||
           month == 9 || month == 11 ) {
        daysInMonth = 30;
      } else {
        if ( (year%4 == 0 && year%100 != 0)
             || (year%400 == 0) ) {
          daysInMonth = 29;
        } else {
          daysInMonth = 28;
        }
      }
    }
    day = day + 1;
    if ( day > daysInMonth ) {
      day = 1;
      month = month + 1;
      if ( month > 12 ) {
        month = 1;
        year = year + 1;
      }
    }
  } // setToNextDay()
} // Date_Beast
```

Figure 4-3: *The Beast*

4.3 Conclusion

Large, monolithic units of code are difficult to understand. Program decomposition into suitable units[1] is important to understanding.

There is no doubt that solutions like the Beauty should be preferred. The Beauty is not only superior in structure, it is also superior from a learning theoretic point of view. Small units reduce cognitive load [9, 25], structural similarities support the recognition of programming plans or patterns [6, 32, 34], and the frequent appearance of mnemonic names help to give meaning to program elements [10, 13].

The essence of developing programming examples is finding an appropriate structure that supports understanding, and hence learning. But when is one structure better than another? And how much better is it? Can we provide a yardstick for measuring the potential understandability of programs?

5. READABILITY AND UNDERSTANDABILITY

A basic prerequisite for understandability is readability. The basic syntactical elements must be easy to spot and easy to recognize. Only then, one can establish relationships between the elements. And only when meaningful relationships can be established, one can make sense of a program. Although readability is a component of understandability in SEI's quality measures taxonomy [29] and there is a large body of literature on software measurement, we couldn't find a single publication on measures for software readability.

5.1 The Flesch Reading Ease Score

The Flesch Reading Ease Score (FRES) is a measure of readability of ordinary text [11, 35]. Based on the average sentence length (*words*/*sentences*) and the average word length (*syllables*/*words*) a formula is constructed to indicate the grade level of a text. Lower values of the ratios indicate easy to read text and higher values indicate more difficult to read text. I.e. the shorter the sentences and words in a text, the easier it is to read.

Please note that FRES does not say anything about understandability. The FRES is just concerned with "parsing" a text. Its understanding depends on further factors, like for example familiarity of the actual words and sentence structure, or reader interest in the text's subject.

Flesch's work was quite influential and has been applied successfully to many kinds of texts. There are also measures for other languages than English.

5.2 A Reading Ease Score for Software

Following the idea of Flesch, we introduce a Software Readability Ease Score (SRES) by interpreting the lexemes of a programming language as syllables, its statements as words, and its units of abstraction as sentences. We could then argue that the smaller the average word length and the average sentence length, the easier it is to recognize relevant units of understanding (so-called "chunks" [9, 15, 24, 25]).

A chunk is a grouping or organization of information, a unit of understanding. Chunking is the process of reorganizing information from many low level "bits" of information into fewer chunks with many "bits" of information [24]. Chunking is an abstraction process that helps us to manage complexity. Since abstraction is a key computing/programming concept [2, 16, 19], proper chunking is highly relevant for the understanding of programming examples.

Clearly, there are other factors influencing program readability, like for example control flow, naming, and how much the students have learned already. We will come back to these factors in our discussion section. For a good overview over code readability issues, see [10].

5.3 Measurement Data

As mentioned above SRES only measure readability (ease of parsing) of a program. Readability is necessary but not suffi-

[1] These units can be declarative, functional, or object-oriented. The Beast could for example be improved significantly without introducing further classes.

cient for understanding a program. Other factors such as the structural and cognitive complexity also influence understanding. If we use cyclomatic complexity (*CC*) [23] as a measure of structural complexity and difficulty (*D*) [17] as a measure of cognitive complexity, and calculate these measures for the Beauty and the Beast, we get the figures as shown in the embedded table.

As indicated by the figures, the SRES measure clearly is in favour of the Beauty. Even more so are the standard measures of cyclomatic complexity and difficulty. According to these, the Beast is in total 3.6 times more difficult to understand than the Beauty.

	Program	
Measure	*Beauty*	*Beast*
SRES	10.3	16.2
CC	3.0	17.0
D	7.9	43.2
Total (Σ)	21.2	76.4

Of course, this is just an example; the programs we measure as well as the measures we apply are more or less randomly chosen among countless options. To expand a bit on the empirical investigations, we have investigated a number of other standard measures, and we have extended the suite of program examples.

The measures we have investigated have been selected for their reported significance in the literature; the selected measures are presented in Table 1.

Selected measures	
Acronym	**Description**
LoC	Total lines of code
SRES	The software reading ease score as described in section 5.2.
$CC_{max}(m)$	Cyclomatic complexity; the number of (statically) distinct paths through a method; should be <10 [23].
D	The difficulty of the program [17]
avgV(*c*)-*avgLOC*(*c*)	Factors of the *Maintainability Index*, a measure with high predictive value for software maintainability [36]. The measures report average values for Volume, *V* (size in terms of the numbers operators and operands [17]), *CC*, and *LoC* per class (*c*).
CC/LoC	Average *CC* per *LoC*; should be ≤0.16 [20].
LoC/m	Average *LoC* per method; should be ≤7 [20].
m/c	Average number of methods per class; should be ≤4 [20].
WMC	Weighted Method Count, a product of the three previous measures; should be ≤5 [20].

Table 1: *Selected measures*

The suite of program examples is extended from two to five representing a continuum of programs solving the Date problem: *Beauty* (E_1), *Good* (E_2), *Bad* (E_3), *Ugly* (E_4), *Beast* (E_5). E_2 is the same as E_1 except that *daysInMonth* is handled by nested if's. E_3 is the same as E_2 except that the classes are not nested. E_4 is the same as E_5 except that *setToNextDay* is decomposed into helper methods.

The result of our investigations is captured in Table 2. For all measures, lower values are considered better. Threshold values suggested in the literature are given in column *T*.

		Program				
Measure	*T*	E_1	E_2	E_3	E_4	E_5
LoC		50	59	57	31	32
SRES	7±2	10.3	8.9	9.3	11.9	16.2
$CC_{max}(m)$	10	3	7	7	7	17
D		7.92	7.15	9.71	22.4	43.2
avgV(c)		387	412	363	752	798
avgCC(c)	10	4.8	6.25	5.25	14.0	18.0
avgLoC(c)		12.5	14.8	14.3	31.0	32.0
CC/LoC	0.16	0.4	0.42	0.37	0.45	0.56
LoC/m	7	2.9	3.27	4.09	6.75	14.0
m/c	4	3.3	3.75	2.75	4.0	2.0
WMC	5	3.6	5.2	4.1	12.2	15.8
	T	E_1	E_2	E_3	E_4	E_5

Table 2: *Values of selected measures for sample programs*

6. DISCUSSION

Although the measures focus on different aspects of a program, it can be noted that they "favour" programs with high degrees of decomposition (E_1–E_4). This is not surprising, since all research in software design and measurement proposes decomposition as a tool to manage complexity. In relation to education it is important to note that a high degree of decomposition also is an advantage from a cognitive point of view.

However, there are many important aspects of understandability not covered by any measure, like for example the choice of names, commenting rate, etc. Any example must furthermore take into account the educational context, i.e. what the students already (are supposed to) know.

7. TOWARD A MEASUREMENT FRAMEWORK

From the discussion above, we conclude that a framework for measuring programming example understandability should consider properties of the *example itself* as well as the *context of its use*. These properties could be divided into the following orthogonal *intra-example* factors:

- *Readability*: Captures how easy a programming text is to read, based on SRES or similar measures.
- *Structural complexity*: Captures the structural properties of a program, based on measures for control flow complexity (cyclomatic complexity), coupling cohesion, etc.
- *Cognitive complexity*: Captures the information contained in a program, based on Halstead's measures or information theory [18].
- *Commenting*: Captures how well the example is commented (excessive use of comments may be a bad thing).
- *Size*: Captures the size of the example, based on a common size measure like LoC.
- *Consistency*: Captures how well the example follows accepted design principles and rules, based on the amount of "code smells".

and the following orthogonal *inter-example* factors related to usage:

- *Presentation*: Captures the degree of conformance to a style guide or standard or the similarity of style with other examples.
- *Progression*: Captures how well the example "fits" with what the students (are supposed to) know.
- *Vocabulary*: Captures the familiarity of the names occurring in the example (could be a sub-factor of progression).

Such a factorization makes it easier to argue about understandability. A perfectly readable and structured program can very well be difficult to understand, when all identifiers are chosen badly, and perfect values for all example factors are no guarantee for an understandable program, when its usage is badly timed.

8. CONCLUSION AND FUTURE WORK

We have shown that many common software measures respect basic cognitive aspects of example programs, in particular cognitive load; all measures we have investigated say that decomposition is good—the more extreme the decomposition, the better. We also propose and discuss a new measure for software readability (SRES). We conclude that all these measure, although useful, lack in their disregard of factors related to the usage of examples. Based on our discussion, we propose a framework for measuring the understandability of programming examples that aims to take such factors into account.

In future research we aim at developing and empirically validating a simple quality measure of example programs by studying a wide variety of examples from textbooks and course material.

9. REFERENCES

[1] ACM Forum "'Hello, World' Gets Mixed Greetings", *Communications of the ACM*, Vol 45(2), 2002, 11-15.

[2] Armstrong, D. "The Quarks of Object-Oriented Development", *Communications of the ACM*, Vol 49(2), 2006, 123-128.

[3] Bansiya, J., Davis, C. G. "A Hierarchical Model for Object-Oriented Design Quality Assessment", IEEE Transactions on Software Engineering, Vol 28(1), 2002, 4-17.

[4] Briand, L., Wüst, J. "Empirical Studies of Quality Models in Object-Oriented Systems". In M. Zelkovitz (ed.) *Advances in Computers*, Academic Press, Vol 56, 2002, 1-46.

[5] Brooks, R. "Towards a Theory of the Comprehension of Computer Programs", *Intl. J. Man-Machine Studies*, Vol 18(6), 1983, 543-554.

[6] Burkhardt, J.-M., Détienne, F., Wiedenbeck, S. "Object-riented Program Comprehension: Effect of Expertise, Task and Phase", *Empirical Software Engineering*, Vol 7, 2002, 115-156.

[7] Cant, S. N., Henderson-Sellers, B., Jeffery, D. R. "Application of cognitive complexity metrics to object-oriented programs, *Journal of Object-Oriented Programming*, Vol 7(4), 1994, 52-63.

[8] Clancy, M. "Misconceptions and attitudes that infere with learning to program". In S. Fincher and M. Petre (eds.) *Computer Science Education Research*. Taylor & Francis, 2004, pp. 85–100.

[9] Clarck, R, Nguyen, F and Sweller, J. *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load*, Pfeiffer, John Wiley & Sons, 2006.

[10] Deimel, L. E., Naveda, J. F. "Reading Computer Programs: Instructor's Guide and Exercises", CMU/SEI-90-EM-3, Software Engineering Institute, 1990.

[11] Flesch, R. "A new readability yardstick", *Journal of Applied Psychology*, Vol 32, 1948, pp. 221-233.

[12] Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[13] Gellenbeck, E. M., Cook, C. R. "An Investigation of Procedure and Variable Names as Beacons During Program Comprehension", *Proc. 4th Workshop on Empirical Studies of Programmers*, 1991, pp. 65-81.

[14] Genero, M., Piattini, M., Calero, C. "A Survey of Metrics for UML Class Diagrams", *Journal of Object Technology*, Vol 4(9), 2005, 59-92.

[15] Gobet, F., Lane, P. C. R., Croker, S., Cheng, P. C. H., Jones, G., Oliver, I., & Pine, J.M. "Chunking Mechanisms in Human Learning" *Trends in Cognitive Sciences*, Vol 5, 236-243.

[16] Görz, G.: "Abstraction as a Fundamental Concept in Teaching Computer Science", *Les langages applicatifs dans l'enseignement de l'informatique*, Specif no. special 93, Rennes/Paris, 1993, 168-178.

[17] Halstead, M. H. "Toward a theoretical basis for estimating programming effort", *Proc of the Annual ACM Conference (ACM/CSC-ER)*, 1975, 222-224.

[18] Khoshgoftaar, T. M., Allen, E. B. "Empirical Assessment of a Software Metric: The Information Content of Operators", *Software Quality Journal*, Vol 9, 2001, 99-112.

[19] Kramer, J. "Abstraction—the key to Computing?" *Communications of the ACM*, to appear.

[20] Lanza, M., Marinesu, R. *Object-Oriented Metrics in Practice*, Springer, 2006.

[21] Li, Y., Yang, H. "Simplicity: A Key Engineering Concept for Program Understanding", *Proc. 9th Internat. Workshop on Program Comprehension*, 2001.

[22] Martin, J. *Principles of Object-Oriented Analysis and Design*, Prentice Hall, 1993.

[23] McCabe, T. J. "A complexity measure", *IEEE Transactions on Software Engineering*, Vol 2(4), 1976, 308–320.

[24] Miller, G.A. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *The Psychological Review*, Vol 63, 1956, pp. 81-97.

[25] Paas, F., Renkl, A. and Sweller, J. "Special Issue on Cognitive Load Theory", *Educational Psychologist*, Vol 38 (1), 2003.

[26] Purao, S., Vaishnavi, V. "Product Metrics for Object-Oriented Systems", *Computing Surveys*, Vol 35(2), 2003, 191-221.

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

[28] Riel, A. *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[29] SEI (Software Engineering Institute) "Quality Measures Taxonomy", http://www.sei.cmu.edu/str/taxonomies/view_ qm.html, Dec 2006, accessed Jan 11, 2007.

[30] Shao, J., Wang, Y. "A new measure of software complexity based on cognitive weights", *Can. J. Elect. Comput. Eng.*, Vol 28(2), 2003, 1-6.

[31] Sweller, J. and Cooper, G.A. "The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra", *Cognition and Instruction*, Vol. 2, pp. 59-89, 1985.

[32] Trafton, J. G., Reiser, B. J. "Studying Examples and Solving Problems: Contributions to Skill Acquisition", Naval HCI Research Lab, Washington, DC, 1992.

[33] Tryggeseth, E. "Support for Understanding in Software Maintenance", PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 1997.

[34] VanLehn, K. "Cognitive Skill Acquisisition", *Annual Review of Psychology*, Vol 47, 1996, 513-539.

[35] Wikipedia "Flesch-Kincaid Readability Test", http://en.wikipedia.org/wiki/Flesch-Kincaid_Readability_ Test, Jan 8, 2007, accessed Jan 12, 2007.

[36] Welker, K. D. "The Software Maintainability Index Revisited", *CrossTalk*, Aug 2001.

[37] Westfall, R. "'Hello, World' Considered Harmful", *Communications of the ACM*, Vol 44(10), 2001, 129-130.